THE UNIVERSITY of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Parallel Parsing of Context-Free Languages on an Array of Processors

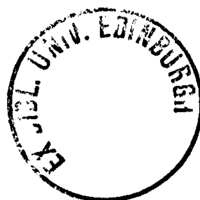
Laurent Chevalier Langlois

submitted for the degree of

Ph D

University of Edinburgh

1988



Abstract

Kosaraju [Kosaraju 69] and independently ten years later, Guibas, Kung and Thompson [Guibas 79] devised an algorithm (K-GKT) for solving on an array of processors a class of dynamic programming problems of which general context-free language (CFL) recognition is a member. I introduce an extension to K-GKT which allows parsing as well as recognition. The basic idea of the extension is to add counters to the processors. These act as pointers to other processors. The extended algorithm consists of three phases which I call the *recognition* phase, the *marking* phase and the *parse output* phase. I first consider the case of unambiguous grammars. I show that in that case, the algorithm has $O(n^2 \log n)$ space complexity and a linear time complexity. To obtain these results I rely on a counter implementation that allows the execution in constant time of each of the operations: set to zero, test if zero, increment by 1 and decrement by 1. I provide a proof of correctness of this implementation. I introduce the concept of efficient grammars. One factor in the multiplicative constant hidden behind the $O(n^2 \log n)$ space complexity measure for the algorithm is related to the number of non-terminals in the (unambiguous) grammar used. I say that a grammar is k-efficient if it allows the processors to store not more than k pointer pairs. I call a 1-efficient grammar an efficient grammar. I show that two properties that I call *nt-disjunction* and *rhs-disjunction* together with unambiguity are sufficient but not necessary conditions for grammar efficiency. I also show that unambiguity itself is not a necessary condition for efficiency. I then consider the case of ambiguous grammars. I present two methods for outputting multiple parses. Both output each parse in linear time. One method has $O(n^3 \log n)$ space complexity while the other has $O(n^2 \log n)$ space complexity. I then address the issue of problem decomposition. I show how part of my extension can be adapted, using a standard technique, to process inputs that would be too large for an array of some fixed size. I then discuss briefly some issues related to implementation. I report on an actual implementation on the I.C.L. DAP. Finally, I show how another systolic CFL parsing algorithm, by Chang, Ibarra and Palis [Chang 87], can be generalized to output parses in preorder and inorder.

Josée Lafontaine and Anne-Marie Brunanchon, of Université de Montréal, helped with the typing of parts of this dissertation. Thank you Josée and Anne-Marie.

During my three years in Edinburgh, I shared an office with Murray Cole (now Dr. Cole) and Martin Illsley. It was good fun to have Murray and Martin in the office to chat (procrastinate). They both gave me much appreciated encouragements. I also benefited from their comments on my work. I also had very interesting discussions with Faron Moller. Faron pointed out to me the existence of a most important algorithm (GKT). Tom Horton (now Dr. Horton) was my Unix, TeX and LaTeX wizard. The good thing about Tom was that he was always in his office. The other good thing is that he was in no hurry to finish his Ph D and so, he was always willing to help me. Eric Wilson once declared: "Friends come before the room." The room in question is what we called the West Room at Eric's house, Seaview. It is a huge room with nine feet high windows overlooking the Firth of Forth. Eric sacrificed having this room as his bedroom so that we (Lise and I) would move into his house. Eric sheltered us for our last year in Edinburgh. That year, with Eric, was our best year. Regina Weinert sheltered me (and Lise as well) during the summer of 1988 (the last months). During that period, I could not have stayed in a better place than in Regina's flat and I could not have had better company than Regina's. Pete Baden, Russell Green and Jo Blishen helped by just being good friends. Thank you very much my friends, thank you Eric, Faron, Jo, Martin, Murray, Pete, Regina, Russell and Tom.

Lise Desjardins deserves a paragraph to herself (and much more). Before, the words "moral support" did not mean much to me but now I am in such a better position to appreciate all of what they stand for. I have leaned often on Lise's support, sometimes quite heavily ("Ph D blues" you understand) and at times, awkwardly. Her deep love and her gentle strength have never faltered. Working for a Ph D is not the most fun experience I have ever gone through. Going through it with Lise by my side contributed much to make it a significantly happier experience and I will always remember the three years we spent together in bonny Scotland with great fondness. On the more practical side of things, I thank Lise for her technical and domestic support. In the last couple of months of the writing, Lise joined me and she helped a great deal with the typing and with some urgent LaTeX hacking. Lise also allowed me to leave to her the carrying out of all the vital life support activities that are the food buying, food cooking and all that... I am most

À ma grande amie, ma petite amie Lise Desjardins.

Table of Contents

List of figures **xi**

List of tables **xiii**

List of abbreviations **xiv**

1. Introduction **1**

1.1 Harnessing Parallelism 1

1.2 Cellular automata, systolic arrays and present day machines 3

1.3 Promises and limitations 4

1.4 Parallelism in parsing 5

1.5 An extension to K-GKT 6

1.6 Overview 8

2. Preliminaries **10**

2.1 CFL parsing 10

2.1.1 The problem (our notation) 10

2.1.2 Chomsky normal form 13

2.2 Parsing of restricted CFLs 17

2.3 Parsing of general CFLs 19

2.3.1 The Cocke-Younger-Kasami algorithm 20

2.3.2 The Earley algorithm 25

2.3.3 The CYK algorithm and dynamic programming 27

2.3.4 The Kosaraju and Guibas, Kung and Thompson array algorithms 29

2.3.5 The extensions of Chiang and Fu 33

2.3.6 The Chang, Ibarra and Palis array algorithm 34

2.3.7 Other work 44

3. An Extension to K-GKT **46**
3.1 Introduction . . . 46
3.1.1 Definitions . . . 49
3.1.2 Desiderata for systolic algorithms . . . 50
3.2 The algorithm . . . 51
3.2.1 The recognition phase (K-GKT plus counters and pointers) . 51
3.2.2 The marking phase . . . 55
3.2.3 The output phase . . . 58
3.2.4 Parse orders . . . 60
3.3 Complexity . . . 62
3.3.1 Space . . . 63
3.3.2 Time . . . 63
3.4 Constant time counters . . . 67
3.4.1 Implementation . . . 67
3.4.2 Proof of correct behavior . . . 68

4. Efficient grammars **78**
4.1 Introduction . . . 78
4.2 G_1 is efficient . . . 79
4.3 NT-disjunction and RHS-disjunction . . . 84
4.3.1 Undecidability of nt/rhs-disjunction . . . 88
4.4 Efficiency, ambiguity and structural ambiguity . . . 91
4.5 A lot of open questions . . . 93

5. Outputting multiple parses **94**
5.1 Introduction . . . 94
5.2 Preliminaries . . . 95
5.2.1 Partitioning/right-hand side pairs . . . 95
5.2.2 Parses and sub-parses . . . 96
5.3 Recursive enumeration: the $O(n^3 \log n)$ method . . . 97
5.3.1 Requesting and providing sub-parses . . . 99
5.3.2 Comments . . . 99
5.3.3 Complexity . . . 101
5.4 Recursive enumeration: the $O(n^2 \log n)$ method . . . 104

5.4.1 The information needed to output the next parse 105
5.4.2 The recognition phases 109
5.4.3 Recording the next pair 111
5.4.4 Identifying the last p/rhs pair 113
5.4.5 The next becomes the current 114
5.4.6 Alternative orderings 114
5.4.7 Complexity 118

6. Problem decomposition **121**

6.1 Introduction 121
6.1.1 The real array 122
6.1.2 The virtual array 123
6.2 The information in auxiliary storage 125
6.2.1 The amount of information 125
6.2.2 The number of values produced by a row of a half-tile 126
6.2.3 The number of values added by a row of a full-tile 127
6.3 Timing issues 128
6.3.1 Doing only useful work 129
6.3.2 The fast belt and slow belt paradox 131
6.4 Initialisation of counters 132
6.5 Recovering the recognition (solution) matrix 134
6.6 Complexity 136
6.6.1 Space 136
6.6.2 Time 137

7. Implementation issues **140**

7.1 Introduction 140
7.2 The CYK combination and parallelism 141
7.3 A look at 3 different machines 143
7.3.1 The Transputer array 143
7.3.2 The Connection Machine 144
7.3.3 The DAP 145
7.4 Pros and cons 147
7.4.1 Computation/communication power balance 147

7.4.2 Code replication 149
7.4.3 The number of processors 150
7.4.4 I/O and problem decomposition 150
7.4.5 A triangular array 151
7.5 An actual implementation 152
7.5.1 An ad hoc parser for $L(G_1)$ 153
7.5.2 DAP Fortran 153
7.5.3 The implementation 156

8. CIP and parse orders 160
8.1 Introduction 160
8.1.1 Valid tree representation 161
8.2 Preorder enumeration 164
8.3 Inorder enumeration 169
8.4 Flattening and the extension to K-GKT 174
8.4.1 Yet more variants 176

9. Discussion 177
9.1 Comparison with other work 177
9.1.1 The extension of Chiang and Fu 177
9.1.2 The CIP algorithm 178
9.2 Simple and active memory 180
9.3 Of practical interest ? 181
9.3.1 Parsing of programs 181
9.3.2 Parsing of natural languages 181
9.3.3 Syntactic pattern recognition and pattern matching 182
9.3.4 Building of optimal search trees 182
9.3.5 Optimal matrix multiplication and optimal file merging . . . 183
9.3.6 A solution in search of a problem 183
9.4 Areas for further research 184
9.4.1 Efficient grammars 184
9.4.2 Problem decomposition 184
9.4.3 CYK combination and PLAs 185
9.4.4 Real performance evaluation 185
9.4.5 CIP 186
9.4.6 Transitive closure 186
9.5 Conclusions 187

A. DAP programs for parsing $L(G_1)$ **189**

A.1 Host component 189

A.2 DAP component 204

Bibliography **241**

List of Figures

2–1 The recognition matrix during recognition of the string (a+a)*a. . . 23

2–2 The CYK algorithm in pseudo-code. 24

2–3 The array of Kosaraju and of Guibas, Kung and Thompson. 30

2–4 The effect of the synchronization strategy. 32

2–5 The parse tree for the CIP algorithm example. 35

2–6 Two equivalent representations of processor contents: a) tabular form; b) *unrolled* form. 36

2–7 The array for the recognition phase of CIP. 37

2–8 The contents of the processors during the recognition phase. 38

2–9 The data exchange occuring between processors during recognition. 39

2–10 The parse extraction sub-array. 41

2–11 The content of the processors during the beginning of the backward and rule laying out phase 43

2–12 The content of the parse extraction sub-array: a) at the end of the rule layout phase; b) at the end of the flattening phase. 44

3–1 The underlying parse tree. 47

3–2 The array after reconfiguration (marking). 48

3–3 Counter initialisation with preset counters. 53

3–4 Counter initialisation with undefined counters. 54

3–5 The marking of a branch by token passing. 57

3–6 State diagram for the output phase. 60

3–7 The output of the parse of a+a. 61

6–1 The real array. 122

6–2 The virtual array. 123

6–3 The output from a half-tile. 129
6–4 The output from a full-tile. 130

7–1 The storage of an 8 bit integer in Host Fortran and DAP Fortran. . 157

8–1 The CIP parse extraction sub-array. 161
8–2 CIP flattening phase adapted for preorder enumeration. 166
8–3 The sub-arrays relative to a tree in the array. 167
8–4 CIP flattening phase adapted for inorder enumeration. 171

List of Tables

3–1 Node operations, tree traversal and parse orders. 62

3–2 Function $\mathcal{F}$. 69

List of abbreviations

A.M.T.	Active Memory Technology Ltd.
CFG	Context-Free Grammar.
CFL	Context-Free Language.
CIP	The Chang, Ibarra and Palis algorithm [Chang 87].
DAP	Distributed Array of Processors.
GHR	The Graham, Harrison and Ruzzo algorithm [Graham 76b].
GKT	Guibas, Kung and Thompson.
I.C.L.	International Computer Limited.
K-GKT	The Kosaraju and Guibas, Kung and Thompson algorithm [Kosaraju 69] [Guibas 79].
PCP	The Post Correspondence Problem.
rhs	Right-hand side (of a rule).

Chapter 1

Introduction

1.1 Harnessing Parallelism

A lot of the research currently conducted in the field of Computer Science relates to parallel processing. A phrase like "to harness parallelism", often found in research papers, is evocative of the great hope, justified or not, that people put in the possibilities of exploiting parallelism in problem solving and of the benefits people expect to obtain from such an exploitation. The interest in parallelism has intensified in recent years but the idea of using concurrency in computing is certainly not new. The first reference on the subject appeared in 1842 in a paper by Menabrea [Menabrea 61] which describes Charles Babbage's Analytical Engine [Babbage 22] [Randell 82]. ENIAC [Hartree 46] [Goldstine 46] [Randell 82], the first general-purpose electronic digital computer, which was operational in 1946, had 25 computing units (20 accumulators, 1 multiplier, 1 divider/square rooter and 3 table look-up units) that could all work in parallel. The quest for parallelism has been pursued on three fronts: hardware, programming methodology and algorithmic. In the last thirty years a wide variety of machine models, either real or theoretical, have

been designed to take advantage of parallelism, some of them more pragmatic then others. Amongst these we find: computers with pipeline architectures, vector computers, processor arrays, data flow machines, graph reduction machines and neural networks. (Vector processing is basically an extension to pipeline processing. This relatively sober idea probably represents the approach that has as yet most successfully capitalized on the use of parallelism. This is no doubt due to the fact that this approach suits so well a wide range of scientific calculation (number crunching) applications.) In parallel with developments on the hardware side, developments came about on the programming methodology side (languages, environments). A few years ago, when vector computers started to become more widely spread, a great deal of work dealt with the vectorization of sequential programs [Higbie 79] [Austin 79] [Paul 75] [Kennedy 79]. In recent years, emphasis has been put on the suitability of various programming styles for exploiting parallelism: object oriented programming (SIMULA [Birtwistle 73], SMALLTALK [Goldberg 83]), functional programming (LISP [McCarthy 60], ML [Harper 86] [Wikstrom 87]) and logic programming (PROLOG [Clocksin 81]). Important work has also been done on a more theoretical level with the study of the semantics of concurrency (CCS [Milner 80], CSP [Hoare 78]). Finally, as well as the research on the hardware and methodology sides, research has also been conducted on the algorithmic side of the exploitation of parallelism. It can be argued that this aspect of the problem is probably the most important of the three since, without parallel algorithms, what use could one make of a parallel machine or of a convenient language for writing parallel program? Although the quest for parallel algorithms has been going on for a long time most of the activity in this area of research has taken place in the last few years. This is due to recent advances in technology (VLSI). The outburst of interest in the area has been evidenced by the appearance in 1984 and 1986 of two important new journals devoted to parallel processing the *Journal of Parallel and Distributed Computing* (Academic Press Inc.) and the *International Journal of Parallel Programming*

(Plenum Publishing Corp.)[1] Parallel algorithms have found applications in a variety of fields. Although the most important of these is no doubt numerical analysis [Heller 78] [Miranker 71] [Poole 74], parallel algorithms have been designed to solve problem in areas as diverse as sorting [Thompson 77], searching [Deker 86], graph theory [Hirschberg 83], computational geometry [Miller 84], topology [Beyer 69], information retrieval [Shaw 80] and memory management [Steele 75].

1.2 Cellular automata, systolic arrays and present day machines

People doing research on algorithms often tend to be theoreticians. When designing algorithms such persons will often rely on some idealistic theoretical model of a machine. Only if the model they are using is near enough to reality can their discoveries one day bear fruit. If such is not the case the risk is that their work might well remain only of purely theoretical interest. For this reason, over-idealistic machine models are liable to have a short life.

The late sixties and early seventies saw the attention of a number of researchers focus on algorithms for a machine model that was then called *cellular automata* or *array automata* [Burks 70] [Codd 68] [Beyer 69] [Cole 69] [Hamacher 68] [Smith 71]. The interest then died out. At the end of the seventies, researchers started to pay attention to a slightly different machine model referred to as *systolic arrays* [Kung 79]. The revival of interest for this sort of machine model has been brought about by the rapid technological progress that had taken place, that was still taking

[1] IJPP actually existed before 1986 but under the name *International Journal of Computer and Information Systems.* As it changed name it also changed its orientation.

place then and that is still taking place today in the field of integrated circuits. Suddenly, the model was nearer to reality. It may be said that in the early and mid eighties, with the appearance on the market of processor arrays, such as the DAP with 4096 processors [Gostick 79] [Reddaway 79] [Parkinson 88] (International Computer Limited and Active Memory Technology), the FPS T series (Floating Point Systems Inc.) with up to 16,384 processors [Frenkel 86] and the Connection Machine (Thinking Machine Corporation) with 65,536 processors [Hillis 85], the model became reality.

1.3 Promises and limitations

Parallelism holds promise. The idea of many processors acting simultaneously and cooperating in the task of solving a problem is very attractive to computer scientists. It is an idea that stimulates imagination and excites curiosity. On the other hands, most will realise that parallelism has its limitations. Not all problems have "parallel solutions". In fact, probably not "many" problems do. In no one's mind will the parallel computer ever replace the good old sequential machine. But what are the promises that parallelism can actually fulfill and where do its limitations lie? We may well never find the exact answer to this question. Nevertheless, it seems obvious that in order to profit as fully as possible from the benefits of parallelism, a lot more needs to be known about the field and this warrants that we continue our efforts in the exploration of the area. More specifically, more research needs to be conducted on the algorithmic side of parallelism. We need to know more about the applicability of parallel processing to problems of various sorts. This thought has motivated me in investigating the adequacy of resorting to parallelism for parsing context-free languages (CFLs).

1.4 Parallelism in parsing

Due to the fact that computer programs have to be parsed, the problem of parsing is very well known to computer scientists. In the sixties sequential solutions to the problem have been studied intensively. Some of the work dealt with general CFLs [Hays 67] [Younger 67] [Kasami 65] [Earley 68] [Schorre 64] [Reynolds 65] but most of it concentrated on the parsing of restricted CFLs: *precedence* languages [Floyd 61] [Wirth 66], LR languages [Knuth 65] and LL languages [Lewis 68] [Korenjak 66] [Wood 70]. Today, we can expect most university graduates in computer science to be more or less familiar with at least one parsing method. Relatively few researchers have worked on parallel parsing (I shall have more to say about this in the next chapter). This is somewhat surprising considering how well the problem is known and how well its sequential solutions are known. Most of the work on parallel parsing has been targeted on the parsing of restricted CFLs. In this dissertation I focus my attention on the parsing of general CFLs.

In 1975, Fischer [Fischer 75] listed in the introduction of his thesis (which was primarily about parallel parsing methods based on precedence grammars) various sequential methods and considered which of those could perhaps be deemed good candidates for parallelisation. He rejected the method developed by Earley [Earley 70] arguing that it was too resource consuming. The rapid progress in technology that has occurred since then may justify us today in adopting a different stance on that question. In this dissertation I report on an extension for parsing to a systolic algorithm that performs CFL recognition. The algorithm can be seen as a parallel version of the Cocke-Younger-Kasami (CYK) algorithm [Hays 67] [Younger 67] [Kasami 65] which itself can be seen as a generalisation of Earley [Graham 76b], the method Fischer rejected. The fact that a chapter in the dissertation (chapter 7) is devoted specifically to issues related to the implemen-

tation of the algorithm on real machines and the fact that I also report in this chapter on an actual implementation of the algorithm offer, if not evidence then at least an indication that research into this kind of algorithm need not be motivated by theoretical interest alone. (Fischer's algorithms were all designed for theoretical machines and he could only either prove them correct or simulate them. I was more fortunate, I had access to the real McCoy.) But one must interpret one's results soberly and I have to point out right away (before building up false expectations) that my implementation was on a rather small scale. More about this in the last chapter (Discussion), where it belongs ...

1.5 An extension to K-GKT

The main contribution reported in the dissertation is an extension to an algorithm for CFL recognition due to Kosaraju [Kosaraju 69] [Kosaraju 75] and also to Guibas, Kung and Thompson (GKT) [Guibas 79]. The original algorithm performs CFL recognition. CFL recognition consists in establishing whether a sentence is part of the language generated by a specific context-free grammar (CFG) or not. My extension allows that we do parsing as well as recognition. Parsing consists in finding how a sentence that is part of the language can actually be generated from the grammar.

General CFL recognition (and parsing) is a member of a class of dynamic programming problems. I shall often refer to this class and I shall denote it by the symbol $\mathcal{C}$. (I characterize the problems of this class in section 2.3.3.) All the problems of $\mathcal{C}$ can be solved using the same algorithmic skeleton. Kosaraju discovered (do you invent algorithms or do you simply discover them?) his algorithm in 1969. He reported it as an algorithm for *array automata* and as a solution for CFL recognition. He seemed to have overlooked the fact that his algorithm could also be

applied to the other problems of $\mathcal{C}$. Independently, ten years later, Guibas, Kung and Thompson rediscovered the algorithm. They reported it as a *systolic algorithm* and they explicitly indicated that it could be adapted for any dynamic problem of $\mathcal{C}$. I refer to the algorithm as the K-GKT algorithm.

(It seems as though very few people are aware that the Kosaraju algorithm and the GKT algorithm are really just two versions of the same algorithm. I have not seen one single mention of this fact in the literature, not even in papers that reference both Kosaraju and GKT!)

The K-GKT algorithm computes what I call a *solution matrix*. It runs on a triangular array of orthogonally interconnected processors. Each processor of the array corresponds to an element of the solution matrix. During the algorithm execution, a processor will compute the value of the element it represents. When applied to CFL recognition, the algorithm computes a solution matrix also. In that particular case I call the matrix more appropriately the *recognition matrix*. This is the same matrix as the matrix computed by the Cocke-Younger-Kasami algorithm. My extension is very simple. I add counters to the processors of the array. At various relevant points during the computation of the solution (recognition) matrix, the values of the counters in some processors are saved (in the processors' local stores). These saved counter values act as pointers to other processors. After the computation of the matrix, I use these pointers to reconfigure the array of processors into a tree of processors. In the context of CFL parsing, the resulting tree of processors corresponds to the parse tree sought. The reconfiguration is done in an original way. Instead of resorting to programmable switches as in [Snyder 82], I have certain processors act as *link node* processors and others as *tree node* processors. Link node processors are used solely as communication links.

Throughout the dissertation, I present my extension in the framework of CFL recognition and CFL parsing. The reader must bear in mind however that the

extension also applies to any of the dynamic programming problems of class $\mathcal{C}$. I also discuss issues that relate specifically to parsing.

1.6 Overview

In the first chapter I cover background material. I first define formally the problem of CFL parsing. I then review some of the work that has been done in parallel parsing. I describe in some detail the CYK and Earley sequential parsing algorithms. I characterize the problems of class $\mathcal{C}$. I also describe existing general CFL parallel parsing algorithms.

In the second chapter, I introduce my extension. I restrict the presentation to the case where the grammar is unambiguous. I show that the extension has, in that case, an $O(n^2 \log n)$ space complexity and an $O(n)$ time complexity. To be able to get the linear time complexity I rely on the fact that the operations on the counters can all be performed in constant time. I suggest a counter implementation that meets the required specifications and I prove its correctness.

In the third chapter, I consider grammar properties that could ensure that the processors of the array will not need to store too much information during the execution of the algorithm. I define an *efficient* grammar as one that will limit to 1 the numbers of pointer pairs any processor might be required to store. I suggest two properties: *nt-disjunction* and *rhs-disjunction*. I show that either of these together with unambiguity is a sufficient but not a necessary condition for grammar efficiency. I provide some decidability results related to these properties.

In the fourth chapter, I consider the case of the use of non ambiguous grammars. I define two algorithms for outputting multiple parses. I show that both algorithms output each parse in linear time, that the first requires $O(n^3 \log n)$ space and the

second requires only $O(n^2 \log n)$ space. The latter result is probably one of the more interesting results reported in this dissertation.

For sake of completeness, in the fifth chapter I address the question of problem decomposition, that is the question of how to handle inputs that are too large for some fixed size array. I show how a standard technique can be applied with my extension and I expose some of the implications this involves.

In the sixth chapter, I take a look at issues related to the implementation of the algorithm on real machines. I consider three machines currently available on the market: the Connection Machine, the DAP and the Transputer. I also report on an actual implementation of the algorithm on the DAP.

In the seventh chapter I show how another parallel parsing algorithm, the Chang, Ibarra and Palis (CIP) algorithm [Chang 87], can be generalised to output parses in various orders.

In the concluding chapter I compare my algorithm with those of others. I also discuss some practicality issues related to the use of my algorithm for CFL parsing and for other dynamic programming problems. I suggest areas for further research.

Chapter 2

Preliminaries

2.1 CFL parsing

The problem of parsing context-free language has been described in the literature over and over again. It is probably one of the best known problems to computer scientists. Nevertheless, for sake of completeness, I shall here describe the problem again. This exercise will also serve as a convenient means for introducing my notation[1].

2.1.1 The problem (our notation)

Context-free languages

An *alphabet* is a set of *symbols*. A *string* over an alphabet Σ is a sequence of symbols taken from Σ. I denote the *empty* string, the string with no symbols by ν (nu). A

[1] I use the same notation as in [Aho 72].

language is a set of strings. Σ^n denotes the language consisting of all the strings over Σ of n symbols. Σ^* (Σ^+) denotes the language consisting of all the strings of zero (one) or more symbols over Σ. (We often find in the literature the use of ϕ^*, where ϕ is the empty alphabet, to denote the empty string.)

A context-free grammar (CFG) is a four-tuple $\langle N, \Sigma, P, S\rangle$ where N is a set of symbols called the *non-terminals*, Σ is a set of symbols called the *terminals*, P is a set of ordered couples consisting of a non-terminal and a string of terminals and non-terminals (a string in $(N \cup \Sigma)^*$) and S is a distinguished symbol in N called the *start* symbol. We call the couples of P the *production rules* or the *rewriting rules* or simply the *rules* of the grammar. To denote the rule (A, α) we use the special notation $A \rightarrow \alpha$. We call the first component of a rule its *left-hand side* (A) and the second its *right-hand side* (α). We sometime denote a set of rules with a common left-hand side $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2, \ldots A \rightarrow \alpha_m$ more conveniently as $A \rightarrow \alpha_1|\alpha_2|\cdots\alpha_m$. The α_i are called the *alternatives* for A.

Convention I adopt the following convention. Terminal symbols will be denoted by small roman letters (a, b, c,...), non-terminals by capital roman letters (A, B, C,...) and strings of terminals and non-terminals by small greek letters (α, β,...).

Derivations and parses

Given a grammar $G = \langle N, \Sigma, P, S\rangle$, we have the following relation between strings of $(N \cup \Sigma)^*$: a string $\alpha A \gamma$ *directly derives* the string $\alpha\beta\gamma$ if the rule $A \rightarrow \beta$ is in P. This relation is denoted by $\Rightarrow$ its transitive closure is denoted by $\stackrel{+}{\Rightarrow}$ and its transitive and reflexive closure is denoted by $\stackrel{*}{\Rightarrow}$. If $\alpha \stackrel{*}{\Rightarrow} \beta$, we say that α *derives* β.

The sequence of strings $\alpha_0, \alpha_1, \ldots \alpha_m$ such that $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \alpha_m$ is called the *derivation* of α_m from α_0. A *leftmost* (*rightmost*) derivation is a derivation

$\alpha_0, \alpha_1, \ldots \alpha_m$ such that each α_i ($i > 0$) is derived from α_{i-1} by replacing the leftmost (rightmost) non-terminal in α_{i-1}. To each leftmost derivation corresponds a unique rightmost derivation.

The language generated by a grammar $G = \langle N, \Sigma, P, S \rangle$, denoted $L(G)$, is the set of strings over Σ that can be derived from the start symbol S. Formally: $L(G) = \{\beta \mid \beta \in \Sigma^*, S \stackrel{+}{\Rightarrow} \beta\}$. The strings in $L(G)$ are called the *sentences* of G. A language that can be generated by a CFG is a context-free language (CFL). Two grammars are *equivalent* if they generate the same language.

A leftmost (rightmost) *parse* for a sentence of G is the sequence of the rules (rule numbers) that are involved in each derivation step of a given leftmost (rightmost) derivation. A *parse tree* (or *derivation tree*) for a sentence of G is a directed tree whose internal nodes are labelled by non-terminals and whose leaves are labelled by terminals. The root of the tree is labelled by the start symbol, the sons of a node are labelled by the symbols of the right-hand side of a rule whose left-hand side is the symbol labelling the father. The frontier of the tree is labelled by the symbols of the string. To each parse tree corresponds a unique leftmost derivation (and hence a unique rightmost derivation, a unique leftmost parse and a unique rightmost parse).

A sentence of G is said to be *ambiguous* if there exists more than one parse tree for it and conversely, it is *unambiguous* if there exists only one. A grammar is unambiguous if all its sentences are unambiguous. A language is unambiguous if at least one unambiguous grammar generates it.

The problem of *recognition* consists in, given a string and a grammar G, deciding if the string belongs to $L(G)$ or not. Assuming it does, the problem of *parsing* consists in providing a parse or a parse tree for it.

2.1.2 Chomsky normal form

Many parsing algorithms I refer to throughout the dissertation work only with grammars that are in Chomsky normal form (CNF) [Chomsky 59] [Aho 72]. A grammar G is in CNF if all its rules have a right-hand side consisting of either two non-terminals or of one terminal. The rule $S \rightarrow \nu$, where S is the start symbol of the grammar (and ν is the empty string), can also be in a CNF grammar. Any grammar G not in CNF can be transformed into an equivalent grammar G' in CNF and so the fact that an algorithm may require that a grammar be in CNF is not a severe restriction.

Aho and Ullman present an algorithm (algorithm 2.12 in [Aho 72]) for transforming grammars not in CNF into grammars in CNF. I refer to this transformation as the *standard transformation* for that purpose. I denote it by T. If G is a grammar, I denote by $T(G)$ the grammar obtained by applying T to G. In this section, I give an upper bound on the relative increase in grammar size that the application of transformation T involves. I also indicate how we can recover the parse of a sentence according to a grammar G from the parse of the sentence according to $T(G)$.

T takes as input a *proper* CFG with no *single production.* A proper CFG is one that is *cycle-free*, that is ν*-free* and that has no *useless symbols.* A grammar G is cycle-free if for no non-terminals A of G do we have $A \stackrel{+}{\Rightarrow} A$. It is ν-free if it has no rule of the form $A \rightarrow \nu$ or it has only one which is $S \rightarrow \nu$, where S is the start symbol of G, and S appears in no right-hand side of rule of G. A non-terminal A of a grammar is a useless symbol if we cannot derive from it a string of terminals. A single production is a rule of the form $A \rightarrow B$ where B is a non-terminal (and so is A). We can transform any CFG that is not proper and/or that contains single productions into a proper CFG with no single production [Aho 72]. Analysing the relative increase in grammar size involved by such a transformation falls outside the

scope of this dissertation. I shall thus simply assume that the grammars we may want to transform using T are proper and contain no single productions. This is a reasonable assumption since most parsing algorithms require grammars with these properties. For convenience to the reader I reproduce below, with slight variations, algorithm 2.12 of [Aho 72] (T).

Transformation T

input: a proper CFG $G = < N, \Sigma, P, S >$ with no single production.

output: an equivalent grammar in CNF $G' = < N', \Sigma, P', S >$.

In the following, X_i' denotes a new non-terminal if X_i is a terminal and X_i' denotes X_i if X_i is a non-terminal. P' is obtained as follows:

1. Add each rule of the form $A \rightarrow a$ in P to P'.
2. Add each rule of the form $A \rightarrow B\,C$ in P to P'.
3. If $S \rightarrow \nu$ is in P, add $S \rightarrow \nu$ to P'.
4. For each rule in P of the form $A \rightarrow X_1\,X_2$ where X_1 or X_2 or both X_1 and X_2 are terminals, add to P' the rule $A \rightarrow X_1'\,X_2'$.
5. For each rule of the form $A \rightarrow X_1 X_2 \ldots X_k$ in P where $k > 2$ add to P' the following set of rules: $A \rightarrow X_1'\,B_1$, $B_1 \rightarrow X_2'\,B_2, \ldots, B_{k-2} \rightarrow X_{k-1}'\,X_k$, where the B_i are new non-terminals.
6. For each new non-terminal A' introduced by the previous two steps, add to P' the rule $A' \rightarrow a$.

N' is obtained by adding to N the new non-terminals introduced by the steps above. $T(G) = G' = < N', \Sigma, P', S >$.

Relative sizes of $T(G)$ and G

I now give an upper bound on the relative sizes of $T(G)$ and G for any proper CFG G with no single productions. I define the ***right-hand side length*** (***rhs-length***) of a grammar G as the sum of the right-hand side length of all the rules in G. I denote this value by rhs-length(G). The upper bound is given by the following theorem.

Theorem 2.1.1 *For any proper CFG $G =< N, \Sigma, P, S >$ with no single productions the following three relations hold:*

1. $\mid N' \mid - \mid N \mid < 2$ *rhs-length*(G)

2. $\mid P' \mid < 3$ *rhs-length*(G)

3. *rhs-length*$(G') < 3$ *rhs-length*(G)

where $G' =< N', \Sigma, P', S >$ is the grammar obtained by applying T to G.

Proof Observe first that the steps in T involving rules of P with right-hand sides of length 2 or less never introduce more new non-terminals than there are symbols in these right-hand sides. Second, observe that the only step involving rule of P with right-hand sides of length greater than 2, step 5, never introduces more than $2k-2$ new non-terminals when dealing with a rule whose right-hand side is of length k. It introduces the $k-2$ non-terminals denoted $B_1, \dots B_{k-2}$ and it can introduce up to k non-terminals denoted X_i'. Since each rule in P is considered by at most one step and since it is considered at most once by all the steps, it follows that relation 1 above is true. Third, observe that any step dealing with a rule $A \rightarrow \alpha$ of P never adds to P' more rules than there are symbols in α. From this observation and from relation 1 and the fact that step 6 may add to P' as many rules as the number of new non-terminals introduced by the transformation it follows that relation 2 is

true. Fourth, observe that step 1 to 4 add to P' rules whose right-hand sides are of the same length as corresponding right-hand sides in P. Fifth, observe that for each rule $A \rightarrow \alpha$ in P considered by step 5, the rhs-length of the set of rules added to P' is always equal to twice the length of α minus 2. Finally, observe that a rule of the form $a' \rightarrow a$ is added to P' in step 6 if and only if the terminal a appears at least once in a rule right-hand side of P. Hence, the rhs-length of all the rules added by step 6 cannot exceed the rhs-length of G. The truth of relation 3 follows from the last three observations. ■

Parse recovery

One may want to parse a sentence using a grammar G', because parsing with G' is easy, and then be able to easily transform this parse into a parse of the same sentence but according to another grammar G equivalent to G'. It could be that the parse according to G is more convenient for some reason but that parsing the sentence with G is difficult.

It can easily be shown that from the leftmost parse π' of a sentence w according to a grammar $T(G) = G' = < N', \Sigma, P', S >$ we can obtain the leftmost parse π of w according to G as follows. Suppose that we number each rule inserted in P' by the steps 1 to 4 using the same number as the number of the rule in P involved in the insertion. Suppose also that we do the same for the first rule in each set of rules inserted in P' by step 5 (i.e. $A \rightarrow X_1' B_1$ gets the same rule number as $A \rightarrow X_1 X_2 \ldots X_k$) and that any other rule inserted in P' by T is numbered differently from every rule in P. π is obtained from π' by deleting any rule number in π' that is not a rule number of G [Aho 72]. The recovery can be done in a time proportional to the length of π'. We can do the equivalent with rightmost parses but only if we apply to G a variant of transformation T. Let us call the latter T'. The difference between T and T' lies in step 5. Instead of inserting in P' set of

rules

$$A \rightarrow X_1' B_2,\ B_2 \rightarrow X_2' B_2, \ldots, B_{k-3} \rightarrow X_{k-2}' B_{k-2},\ B_{k-2} \rightarrow X_{k-1}' X_k'$$

for a rule of the form $A \rightarrow X_1 X_2 \ldots X_k$ in P the transformation T' inserts the rules[2]

$$A \rightarrow B_1 X_k',\ B_1 \rightarrow B_2 X_{k-1}', \ldots, B_{k-3} \rightarrow B_{k-2} X_3',\ B_{k-2} \rightarrow X_1' X_2'.$$

The rule $A \rightarrow B_1 X_k'$ is the one that gets numbered by the same number as the rule $A \rightarrow X_1 X_2 \ldots X_k$. The other rules must get rule numbers different from every rule number of G.

2.2 Parsing of restricted CFLs

We can divide parsing methods into two broad categories: those applicable to restricted CFLs and those applicable to general CFLs. Most of the work in the area of parsing has been carried out on methods of the former category. This is true for sequential as well as for parallel parsing. In this thesis, I am concerned with methods of the latter category. For the sake of completeness however, I briefly cover here the major research conducted on parallel parsing methods for restricted CFLs.

The work done in this area involved adapting existing sequential parsing methods for use in parallel environments. Fischer [Fischer 75] [Fischer 80] adapted a whole series of precedence parsing techniques for their implementation on vector computers. The skeleton of his algorithms is as follows. The input is first stored

[2]The set of rules does not strictly have to be exactly as indicated but it must contain the rule $A \rightarrow B_1 X_k'$.

as a vector of tokens. Then, from this vector, the vector of all the precedence relations is computed. This last vector serves to identify *simple phrases* in the string, i.e. the sub-strings that can be reduced to non-terminals. The reductions of all the detected simple phrases is effected, the resultant vector is compressed and the whole process is repeated until a single element vector containing the start symbol is obtained. For a particular type of grammar which Fischer calls *arithmetic infix grammars*, Fischer also developed techniques to carry out semantic analysis and code generation as well as parsing. Fischer [Fischer 75] and, a few years later, Schell [Schell 79], extended the LR method for use on a linear array of processors. In their methods, each processor of the array is assigned a segment of the input which it processes in an LR fashion. This implies that processors have to deal with unknown left-contexts. Fischer tackled this problem by having the processors handle multiple stacks while Schell resorted to a *super-initial* state, an LR state meaning *I can be anywhere in any production.* Fischer concentrated on parsing while Schell also addressed the problem of error recovery and semantic analysis. For the former, he adapted the method of Mickunas and Modry [Mickunas 78] and for the latter, he investigated the parallel evaluation of attributed parse trees [Knuth 68]. Fischer and Schell showed that their methods have linear time complexity. (This is a worst case complexity measure. Schell mentioned that in the best case, the time complexity is $O(\log n)$.) Asymptotically, this offers no improvement over the sequential methods. Fischer showed however that in real terms, his parallel LR parsing method is significantly faster than the sequential one. He simulated his method, measured its efficiency and came up with speed-up factors of between 4 and 8.

Frank [Frank 79] is another researcher who worked on adapting the LR method for a parallel environment. He also worked on the LL method. Fischer and Schell resorted to parallelism to improve the efficiency of the LR method, Frank used it to enlarge the classes of languages the LR and LL methods could parse. His

scheme allowed for a bounded degree of non-determinism in the languages. All the processors scanned the input simultaneously. The multiplicity of processors served to cover the multiplicity of paths in the tree of possible computations. The degree of non-determinism of the parsable languages was bounded by the number of processors available.

Other researchers have worked on parallel parsing of restricted CFLs. I shall not mention them all here. Among them we find: Lipkie [Lipkie 79], Lincoln [Lincoln 70], and Zosel [Zosel 73].

2.3 Parsing of general CFLs

Very little work has been done on parsing of general context-free languages. The few methods that have been suggested are based on two (sequential) algorithms: the Cocke-Younger-Kasami (CYK) and Earley algorithms. Kosaraju [Kosaraju 69] [Kosaraju 75] and independently, ten years later, Guibas, Kung and Thompson (GKT) [Guibas 79], devised a clever array algorithm for CFL recognition (K-GKT). The array algorithm is actually applicable to a wide class of dynamic programming problems (class $\mathcal{C}$, see section 2.3.3) of which CFL recognition is just one member. The algorithm allows a direct parallel implementation of the CYK algorithm. Up until a few of years ago, the research conducted looked only at CFL recognition and paid no attention to parsing. In 1984, Chiang and Fu [Chiang 84], and in 1986, Chang, Ibarra and Palis [Chang 87] reported on results concerning both aspects of the problem. Chiang and Fu resorted to the K-GKT algorithm to implement a weakened version of Earley's algorithm. They also suggested a scheme for the extraction of the parse of the input from the array and a scheme for the detection of errors. Ibarra, Kim and Palis [Ibarra 86] devised a wholly new algorithm for the computation of the recognition matrix. Chang, Ibarra and Palis [Chang 87]

extended this algorithm so as to allow the computation to be traced backward and for the parse leading to the successful recognition to be recovered in the process. I shall say more about this work later in this chapter.

2.3.1 The Cocke-Younger-Kasami algorithm

The CYK algorithm uses a grammar in Chomsky normal form (CNF). As was mentioned in section 2.1.2, any CFG can be transformed into an equivalent CFG in CNF so no generality is lost by this requirement. (There exists a generalised version of the algorithm that will accept grammars of any form.) The algorithm is composed of two phases. The first one computes a matrix called the *recognition matrix* and the second one generates a parse for the string from the information contained in the matrix.

CYK combination To describe the first phase, it is useful here to define, given a grammar $G = \langle N, \Sigma, P, S\rangle$, an operation I shall call the *CYK combination.* The CYK combination takes as argument an ordered pair of sets of non-terminals from N. It returns the set of all the non-terminals which appear on the left-hand side of rules whose first non-terminal of the right-hand side is in the first set of the pair and whose second non-terminal is in the second set. Formally:

$$\text{CYK combination}(S_1, S_2) = \{A \mid A \rightarrow B\,C \in P, B \in S_1, C \in S_2\}$$

For example, say we have the following grammar:

$$G_0 = \langle\{A, B, C, D, E, F\}, \{g, h, i\}, P_0, A\rangle$$

where P_0 is:

$$\{A \rightarrow A\,D\,,\ A \rightarrow A\,E\,,\ C \rightarrow A\,B\,,\ D \rightarrow F\,C\,,\ E \rightarrow B\,E\,,\ B \rightarrow g\,,\ E \rightarrow h\,,\ F \rightarrow i\}$$

and say we have the two sets $S_1 = \{A, B, C\}$ and $S_2 = \{D, E, F\}$. The CYK combination of (S_1, S_2) results in the set $\{A, E\}$ and the CYK combination of (S_2, S_1) yields the set $\{D\}$.

The computation of the recognition matrix

For an input string of length n (n tokens), the recognition matrix M consists of an $n \times n$ upper-triangular matrix. The entries of the matrix are sets of non-terminals. A non-terminal will be a member of an entry $M_{i,j}$ if and only if the portion of the input string starting at position i and ending at position j can be derived from that non-terminal. I call this sub-string the sub-string (or the string) *spanned* by the entry. The algorithm first initialises the entries on the diagonal. This is done by inserting into each entry every non-terminal that directly derives the terminal in the string's corresponding position. The algorithm then proceeds by computing the entries on the line next to the diagonal and then those on the next line and so on until the whole matrix has been computed. The value of an entry $M_{i,j}$ not on the diagonal is computed from the values of the pairs of entries $(M_{i,k}, M_{k+1,j})$, for $k = i \ldots j-1$. We set the entry $M_{i,j}$ to the value of the union of the results of the application of the CYK combination on these pairs. Each pair corresponds to one partitioning of the sub-string associated with the entry $M_{i,j}$. Let's suppose that for a given k and a given rule $A \rightarrow BC$ that B is in $M_{i,k}$ and C is in $M_{k+1,j}$. Then $B \stackrel{*}{\Rightarrow} a_i, \ldots a_k$, $C \stackrel{*}{\Rightarrow} a_{k+1}, \ldots a_j$ and hence $A \stackrel{*}{\Rightarrow} a_i, \ldots a_j$. This fact is reflected by the insertion of A in $M_{i,j}$ via the CYK combination of $M_{i,k}$ and $M_{k+1,j}$. If at the end of the matrix computation, the start symbol is a member of $M_{1,n}$ the symbol derives the whole input string and the input is accepted. In such a case we can proceed with the generation of the parse using the information of the recognition matrix.

example:

Say we have the following grammar:

$$G_1 = \langle\ \{E, T, F, E_{)}, {}_{+}T, {}_{*}F, \underline{(}, \underline{)}, \underline{+}, \underline{*}\}\ ,\ \{a, (,), +, *\}\ ,\ P\ ,\ E\ \rangle$$

where P is:

(1)	$E \rightarrow E\ \ {}_{+}T$	(7)	$T \rightarrow \underline{(}\ \ E_{)}$	(13)	$\underline{(} \rightarrow ($
(2)	$E \rightarrow T\ \ {}_{*}F$	(8)	$T \rightarrow a$	(14)	$\underline{)} \rightarrow)$
(3)	$E \rightarrow \underline{(}\ \ E_{)}$	(9)	${}_{+}T \rightarrow \underline{+}\ \ T$	(15)	$\underline{+} \rightarrow +$
(4)	$E \rightarrow a$	(10)	$F \rightarrow \underline{(}\ \ E_{)}$	(16)	$\underline{*} \rightarrow *$
(5)	$E_{)} \rightarrow E\ \ \underline{)}$	(11)	$F \rightarrow a$		
(6)	$T \rightarrow T\ \ {}_{*}F$	(12)	${}_{*}F \rightarrow \underline{*}\ \ F$		

and say we have the string (a+a)*a, figure 2–1 shows the recognition matrix at various stages of its computation. At the end of the computation, E, the start symbol of G_1, is a member of $M_{1,7}$ so we may conclude that (a+a)*a is in the language generated by G_1.

Generation of a parse

The generation of a parse is a recursive process that takes as input the indices (i, j) of a matrix entry and a symbol A and produces on output a parse corresponding to one derivation of the symbol A into the string $a_i, \ldots a_j$. In the case that $i = j$, the process simply outputs the rule (rule number of) $A \rightarrow a_i$. In the case that $i < j$, the process first finds a pair of entries $(M_{i,k}, M_{k+1,j})$ for which there exists a rule $A \rightarrow B\,C$ such that $B \in M_{i,k}$ and $C \in M_{k+1,j}$. Say we want the parse in leftmost order. The process then outputs the rule (or rule number of) $A \rightarrow B\,C$ and calls itself first with indices (i, k) and symbol B and then with $(k + 1, j)$ and

$$\begin{pmatrix} \{\underline{(}\} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ & \{E,T,F\} & \cdot & \cdot & \cdot & \cdot & \cdot \\ & & \{\underline{+}\} & \cdot & \cdot & \cdot & \cdot \\ & & & \{E,T,F\} & \cdot & \cdot & \cdot \\ & & & & \{\underline{)}\} & \cdot & \cdot \\ & & & & & \{\underline{*}\} & \cdot \\ & & & & & & \{E,T,F\} \end{pmatrix} \quad \begin{pmatrix} \{\underline{(}\} & \emptyset & \emptyset & \emptyset & \cdot & \cdot & \cdot \\ & \{E,T,F\} & \emptyset & \{E\} & \{E_{)}\} & \cdot & \cdot \\ & & \{\underline{+}\} & \{{}_{+}T\} & \emptyset & \emptyset & \cdot \\ & & & \{E,T,F\} & \{E_{)}\} & \emptyset & \emptyset \\ & & & & \{\underline{)}\} & \emptyset & \emptyset \\ & & & & & \{\underline{*}\} & \{{}_{*}F\} \\ & & & & & & \{E,T,F\} \end{pmatrix}$$

a) b)

$$\begin{pmatrix} \{\underline{(}\} & \emptyset & \emptyset & \emptyset & \{E,T,F\} & \emptyset & \{E,T\} \\ & \{E,T,F\} & \emptyset & \{E\} & \{E_{)}\} & \emptyset & \emptyset \\ & & \{\underline{+}\} & \{{}_{+}T\} & \emptyset & \emptyset & \emptyset \\ & & & \{E,T,F\} & \{E_{)}\} & \emptyset & \emptyset \\ & & & & \{\underline{)}\} & \emptyset & \emptyset \\ & & & & & \{\underline{*}\} & \{{}_{*}F\} \\ & & & & & & \{E,T,F\} \end{pmatrix}$$

c)

Figure 2–1: The recognition matrix during recognition of the string **(a+a)*a**.

symbol C. To generate the parse of the input string, we call the process with the indices $(1, n)$ and the start symbol.

Figure 2–2 gives a description in pseudo-code of the two phases of the Cocke-Younger-Kasami algorithm.

Complexity

The recognition phase requires $O(n^3)$ execution time and the parse generation requires $O(n^2)$ execution time [Aho 72] [Younger 67]. We can reduce the execution time for parse generation if, during the recognition, for each entry, we save pointers to the pairs of entries responsible for the insertion of non-terminals in that entry. This allows us to *find a pair of entries* in constant rather than linear time. The space complexity of the algorithm is determined by the size of the matrix. The

```
procedure Compute Matrix;
    for i := 1 to n do M_{i,i} := {A | A → a_i};
    for l := 1 to n − 1 do
        for i := 1 to n − l do
            j := i + l;
            M_{i,j} := ∅;
            for k := i to j − 1 do
                M_{i,j} := M_{i,j} ∪ CYK combination(M_{i,k}, M_{k+1,j});
end

procedure Generate Parse ((i, j), A);
    Find a pair of entries (M_{i,i+k}, M_{i+k,j}) such that
        for a A → BC ∈ P, B ∈ M_{i,i+k} and C ∈ M_{i+k,j};
    Output the rule A → BC ;
    call Generate Parse ((i, i + k), B);
    call Generate Parse ((i + k, j), C);
end
```

Figure 2–2: The CYK algorithm in pseudo-code.

matrix is composed of $(n^2 - n)/2$ entries. If we do not save pointers in the entries, each entry needs just enough storage to hold the set of every non-terminals in the grammar. This is a fixed amount independent of the input size and in this case, the space complexity is $O(n^2)$ [Aho 72]. If we do save pointers, we must add a factor of $\log n$ to this measure. If our grammar is ambiguous and we want to obtain all the parses of the input strings then each entry might need to hold a number of pointers proportional to the length of the sub-string it spans. This adds a factor of n and results in a space complexity $O(n^3 \log n)$.

The complexity measures mentioned above are for the straightforward implementation of the CYK algorithm. With much ingenuity, Valiant [Valiant 75] has devised a slightly faster recognition algorithm based on the CYK idea. He showed how the computation of the recognition matrix reduces to computing the transitive closure of a boolean matrix which in turns reduces to boolean matrix multiplication. Resorting to Strassen's algorithm for matrix multiplication [Strassen 69], he devised a method for CFL recognition that takes $O(n^{2.81\cdots})$ time[3]. This is the fastest offline sequential CFL recognition algorithm known today. The algorithm is said to be *offline* because it requires to have the whole input available from the start.

2.3.2 The Earley algorithm

The work reported in this thesis is based on the CYK algorithm. However, I need to say a few words here about another sequential parsing algorithm for general CFLs, the Earley algorithm [Earley 68] [Earley 70]. As we shall see later on, other researchers in the field of parallel parsing have mentioned this algorithm. Unlike CYK, the Earley algorithm can work with grammars of any form (not necessarily in CNF). Like CYK, the algorithm first computes a *recognition matrix.* (In the original description of the algorithm [Earley 70], the data structure consisted of lists rather than a matrix. I refer here to a version due to Graham, Harrison and Ruzzo [Graham 76b].) The entries of the matrix are sets of *dotted rules* instead of sets of non-terminals. A dotted rule is a grammar rule whose right-hand side is divided in two parts. We denote such a rule as follows: $A \rightarrow \alpha \cdot \beta$. We call the first part of the right-hand side, the *scanned portion* of the rule (α) and the second

[3]The hidden constant in this complexity measure is so large that we can consider the method unsuitable for any practical application.

part, its *remainder* (β). In Earley, an entry $M_{i,j}$ will contain dotted rules whose scanned portions derive the string $a_i \cdots a_j$. It will not contain all such rules. For a dotted rule $A \rightarrow \alpha \cdot \beta$ to be inserted in $M_{i,j}$, the following condition also has to be satisfied: $S \stackrel{*}{\Rightarrow} a_1 \cdots a_{i-1} A \rho$ for some $\rho \in \Sigma^*$. In other words, a dotted rule can be inserted in an entry only if it is consistent with the context to the left of the sub-string spanned by the entry. This is where the main difference between Earley and CYK lies. Graham, Harrison and Ruzzo modified the Earley algorithm and obtained a *weakened* version of the algorithm in which the second condition just mentioned is dropped. They pointed out that this weakened Earley algorithm is essentially the same as a generalised version of CYK. I shall not go into the details of the Earley algorithm. Descriptions of differing flavours can be found in [Earley 68] [Earley 70] [Aho 72] [Graham 76b] [Winograd 83] (Winograd indirectly describes Earley's algorithm when describing the *Chart parser*[4]).

Complexity

Assuming we are only interested in one parse for the input string, the time complexity of the algorithm, as CYK, is $O(n^3)$ for ambiguous grammars. For unambiguous grammars, Earley is faster then CYK and takes $O(n^2)$ time. The space complexity is of $O(n^2)$ if we do not save pointers in matrix entries and of $O(n^2 \log n)$ otherwise [Aho 72][5].

---

[4]A *Chart parser* is a parser based on a data structure, called a *Chart*, which consists of a set of vertices, one for each position between tokens of the input string, and a set of labelled directed edges joining pairs of vertices.

[5]In their book, Aho and Ullman consider pointers to be of fixed size and so, the space complexity measures they provide for CYK and Earley do not include any $\log n$ factor.

The Graham, Harrison and Ruzzo variant

Graham, Harrison and Ruzzo [Graham 76b] proposed a more efficient variant of the original recognition phase in Earley. Their variant (GHR) implies a rearrangement in the order of the operations and a pre-computation of tables from the grammar. They were able to implement their algorithm in a way similar to which Valiant implemented CYK. Instead of resorting to Strassen's algorithm, they resorted to the boolean matrix transitive closure algorithm of Arlazarov, Dinic, Kronrod and Faradzev [Arlazarov 70] [Aho 74]. They obtained a CFL recognizer with time complexity $O(n^3/\log n)$. This is slower then Valiant's recognizer. It has the advantage however of being *online*. An online algorithm is an algorithm which can read the input while executing, without knowing in advance what the size of the input will be. GHR is the fastest online sequential CFL recognition algorithm known today.

2.3.3 The CYK algorithm and dynamic programming

CFL recognition is a member of a large class of dynamic programming problems that can all be solved using a common algorithmic skeleton. I denote this class by the symbol $\mathcal{C}$. $\mathcal{C}$ contains problems such as: the building of optimal binary search trees; file merging; computation of order statistics, string matching and the optimal multiplication of matrices. An instance of a problem of $\mathcal{C}$ consists of a sequence of some objects $b_1 b_2 \cdots b_n$. We can partition a sequence of n objects in $n-1$ different ways. Each partitioning yields two sub-sequences that are themselves instances of the original problem. To solve the problem we need only to consider the pairs of solutions for the sub-sequences of the $n-1$ partitionings of the sequence. The dynamic programming approach to the problem is to work bottom up. We start first by finding the solutions for all the sub-sequences of length one, then those for the sub-sequences of length two and so on. An implementation of this strategy consists in the computation of a *solution matrix* M (corresponding to

the recognition matrix in CYK). At the end of the matrix computation, an entry $M_{i,j}$ will contain the solution for the sequence of objects $b_i \cdots b_j$. This entry will have been computed from the information in the pairs of entries $(M_{i,k}, M_{k+1,j})$ for $k = i \ldots j - i$. This general skeleton of the algorithm for computing a solution matrix applies to all the problems of class $\mathcal{C}$. What basically varies from problem to problem is the specific information contained in the matrix entries, the entries' initial values and how we combine the information in the pairs of entries involved in the computation of an entry. For example, in the case of CFL recognition, the entries consist of sets of non-terminals, their initial value is the empty set and the combination is the union of the results of CYK combination over the pairs of entries. The case of the building of an optimal binary search tree is slightly more complicated. An entry $M_{i,j}$ will give the cost of the minimal cost search tree for the set of objects $b_i \cdots b_j$. Objects have *weights* related to the probability that search operations will be performed for them. An entry's initial value will be the sum of the weights of the objects of its set. The final value will be obtained by selecting among the relevant pairs of entries the one whose sum of final values (minimal cost) is minimal and by adding this sum to the entry's initial value (sum of weights). (For details, see [Aho 74].)

Knuth [Knuth 71a] has shown that in the specific case of the building of optimal search trees, the computation of the solution matrix can actually be done in $O(n^2)$ time. Such an improvement is made possible from the observation that for this problem not all partitionings of a sequence s need be considered when searching for the sequence's optimal solution. Let us call s_l^- the sequence s minus its leftmost element and s_r^- the sequence s minus its rightmost element. The search space for the optimal solution of the sequence s can be limited to those partitionings whose left partition is greater or equal to the left partition of s_l^- that led to the optimal solution of s_l^- and whose right partition is greater or equal to the right partition of s_r^- that led to the optimal solution of s_r^-.

As we just saw, solving an instance of a problem of $\mathcal{C}$ involves partitioning the instance (and subpartitioning it and so on) in the right way. We can describe this partitioning as a binary tree. Each node of the tree corresponds to a portion of the instance and its two subtrees correspond to the left and right partitions of this portion. Let's call this tree the *solution tree.* The solution matrix will state if a solution exists or it will tell us what the cost of the optimal solution is. Often, we will also need the solution tree. From the information in the solution matrix, we can reconstitute the solution tree. The second phase of the CYK algorithm does just that, it generates a parse which is just a flattened representation of the solution tree called, in this instance, the parse tree. In the case of the building of an optimal search tree, the solution tree will consist of the search tree sought.

2.3.4 The Kosaraju and Guibas, Kung and Thompson array algorithms

Kosaraju[Kosaraju 69] [Kosaraju 75] and Guibas, Kung and Thompson (GKT) [Guibas 79] came up with a very clever processor array algorithm for the computation of the dynamic programming solution matrix mentioned above. Kosaraju presented his algorithm in the context of CFL recognition while Guibas, Kung and Thompson presented theirs in the more general context of $\mathcal{C}$. Kosaraju referred to his algorithm as an *array automaton.* GKT used instead the more modern term *systolic array.* The two versions of the algorithm differ only very slightly. I shall point out their difference later.

The algorithm operates on a triangular array of orthogonally connected processors. We can think of the array as a representation for the solution matrix. Each processor of the array corresponds to an entry of the matrix. During execution, a processor computes the value of the entry it represents. (For short, I will call this value the value of the processor.) I shall index the processors in the same way

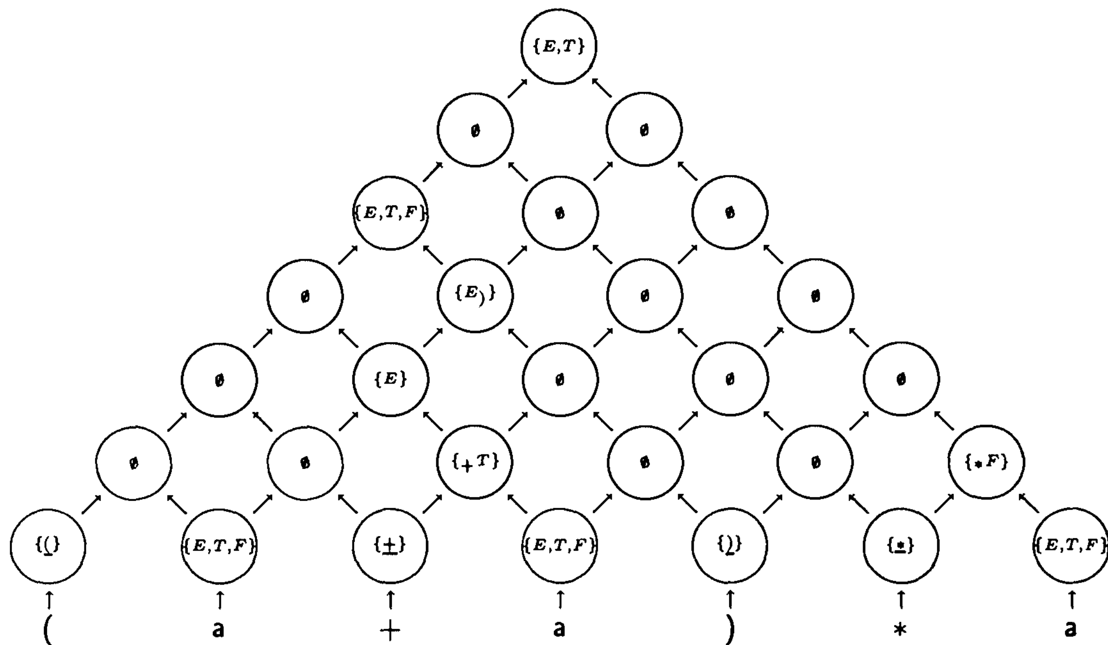


Figure 2–3: The array of Kosaraju and of Guibas, Kung and Thompson.

as we index the matrix entry. The processors are connected to their four neighbours in the same row and column. The values of the processors on the ***diagonal*** $(\{P_{i,i}\}, i = 1 \ldots n)$ are set at initialisation. To compute its value, a processor $P_{i,j}$ not on the diagonal $(i < j)$ will need the values of the processors on its left, $P_{i,k}$ for $k = i \ldots j-1$, and of those below, $P_{k,j}$ for $k = i+1 \ldots j$. These values will be handed to $P_{i,j}$ by its left and bottom neighbours, $P_{i,j-1}$ and $P_{i+1,j}$. The processors pass on to the right (top) the values they receive from the left (bottom) to allow these to reach the other processors that will need them. For the same reason, once a processor has computed its value, it sends it to both its right and top neighbours. Figure 2–3 (page 30) shows a graphical representation of the array applied to the recognition example introduced earlier. For reasons that will prove obvious later, I have tilted the array by 45° counter clockwise.

One of the main ideas behind a systolic network is to have the data circulate in such a way that they arrive at the right place at the right time. Notice that in the

case considered here, a processor $P_{i,j}$ will need to hold simultaneously the values of its *furthest* left neighbour, $P_{i,i}$, and of its *nearest* lower neighbour, $P_{i+1,j}$, and vice-versa. It will also need to hold simultaneously the values from its second furthest left neighbour and second nearest bottom neighbour and so on. To meet these requirements, the algorithm has the data circulate in each direction on two different channels which I will call, as in GKT's paper, the *fast belt* and the *slow belt.*[6] The algorithm arranges for the values on the fast belt to end up in the reverse of the order in which the processors are laid out and for those on the slow belt to end in the same order, thus allowing processors to receive values from nearby neighbours (via the fast belt) at the same time as they receive values from distant neighbours (via the slow belt). The details are as follows. Let us define the time unit as the time from the beginning of one data transfer to the next (the *beat*). I refer to time zero as the time when all the processors on the diagonal have been initialised. Data travel twice as fast on the fast belt as they do on the slow belt. On the fast belt, they go from one processor to the next on every beat. On the slow one, the data go through a pipeline of two registers in each processor and they go from one processor to the next on every two beats. For a processor $P_{i,j}$, I call the value $j-i$ its *distance* from the diagonal. Processors at distance d from the diagonal compute their values from time $\lfloor 3/2\ d \rfloor$ to $2d-1$ and at time $2d$, they deposit their value onto the fast belts. At time $3/2\ d$ processors at an even distance d from the diagonal transfer the values then on their fast belts onto their slow belts. To compute their value, the processors combine the values coming from the left on their fast belt with those coming from below on their slow belt and vice-versa.

The only difference between the version of Kosaraju and that of GKT is the

[6] I speak of two channels but in fact, in an actual circuit, it could very well be the case that only one channel would connect pairs of processors. In that case, the two channels I refer to would be virtual channels implemented on the real one by multiplexing in time.

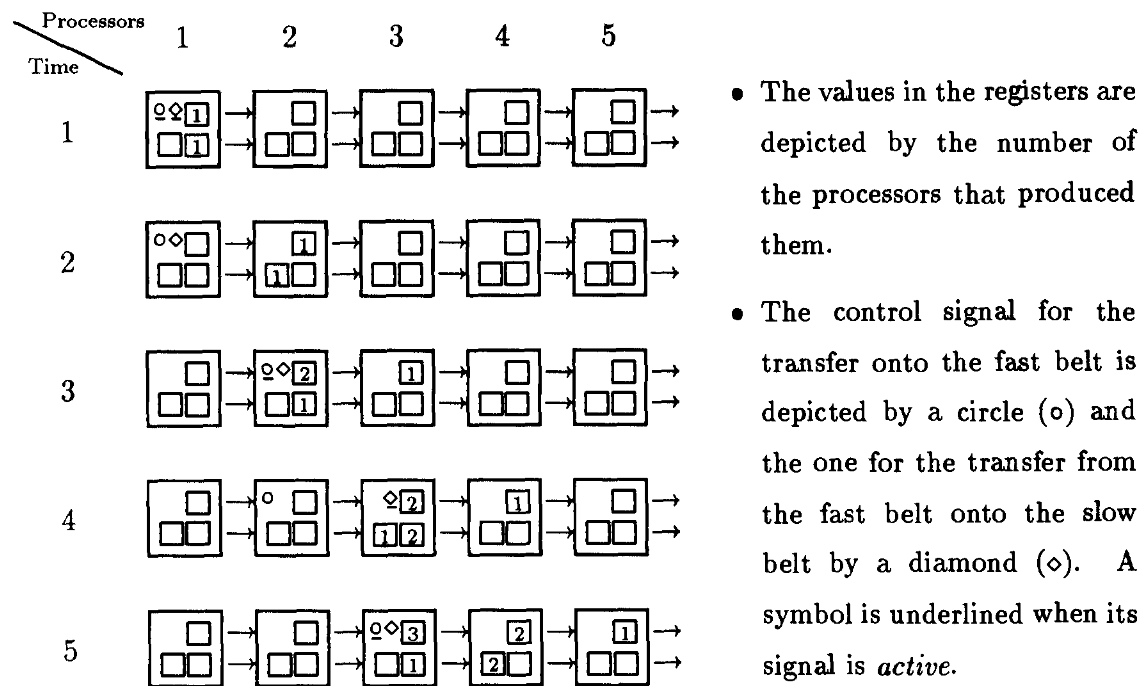


Figure 2–4: The effect of the synchronization strategy.

way in which in each implementation, the processors determine when to put on the fast belts the value they have been computing and when to transfer the value from the fast to the slow belts. Kosaraju resorts to *defined/undefined* flags carried by the values on the channels. For example, if a processor receives for the first time a *defined* value on the fast belt while the slow belt still brings in an *undefined* value, it copies to the slow belt the value on the fast belt and on the next beat, it sends out undefined values on both belts. Guibas, Kung and Thompson on their part have recourse to explicit control signals to trigger the transfer operations. The signals all leave the processors on the diagonal on time 0. The signals to control the transfer of the computed value onto the fast belts travel from one processor to the next on every alternate beat while those to control the transfer from fast to slow belt travel from a processor to its next-but-one on every third beat. Figure 2–4 shows the effect of the synchronization strategy (implemented with explicit control signals).

For a problem instance of length n, an array of $(n^2+n)/2$ processors is needed. This array will compute the solution matrix in $2n$ beats, i.e. in linear time. On a sequential machine, the computation of the solution matrix takes $O(n^3)$ time. The array thus provides a speedup of $O(n^2)$. Since the array consists of $O(n^2)$ processors, the speedup is asymptotically optimal.

2.3.5 The extensions of Chiang and Fu

Chiang and Fu [Chiang 84] resorted to the K-GKT algorithm to implement the recognition phase of the weakened Earley algorithm. As I mentioned earlier, weakened Earley corresponds exactly to a generalised version of CYK. They motivate their recourse to weakened Earley instead of CYK by argumenting that the former avoids the significant overhead incurred by the transformation of a grammar not in CNF to one in CNF. This claim is refutable however. Their algorithm involves, as in GHR, pre-computations of tables from the grammar. Graham, Harrison and Ruzzo [Graham 76b] pointed out that these pre-computations correspond closely to the standard transformation of a grammar to Chomsky normal form.

They suggested two extensions to the basic algorithm, one for extracting the parse (or the parses) of the input and the other for compiling *error-counts*.

Their scheme for parse extraction works in a bottom-up fashion in parallel with the recognition phase. The processors compute in addition to the sets of dotted rules, sets of *parse symbols*. A parse symbol is a pair consisting of a non-terminal and a sequence of rule numbers. A parse symbol (A, π) will be inserted in the set of processor $P_{i,j}$ only if the string of tokens $a_i \cdots a_j$ can be derived from the non-terminal A using the sequence of rules denoted by π. At the end of the algorithm's execution, the second component of any symbol in processor $P_{1,n}$ whose first component is the start symbol denotes a parse for the input. The major drawback of the scheme is that the amount of information the processors

have to store, process and exchange (on each beat) is dependent on the input length and can become exceedingly large. The length of the rule sequence in any parse symbol a processor will record is proportional to the length of the string this processor spans. As a consequence, processors far away from the diagonal will need to hold and exchange more information than processors near the diagonal. For unambiguous grammars, the maximum number of parse symbols a processor might need to store is fixed by the grammar and independent of the input size. For ambiguous grammars, that number, in the worse case, can grow exponentially with the input size [Graham 76a].

Chiang and Fu also presented an extension for the compilation of *error-counts*. An error-count is associated with each dotted rule in the processors' sets. Suppose we have a dotted rule $A \rightarrow \alpha \cdot \beta$ in the set of $P_{i,j}$. Its error-count indicates the minimal number of *deletions*, *insertions* and *substitutions* that it would be necessary to perform on a string derived from α so as to transform it into the string $a_i \cdots a_j$. For each type of error, the number of errors the algorithm can take into account is limited to the number of tokens in the target string, i.e. to $j-i+1$. Different weights can be assigned to each type of error. As with the parse extraction, the algorithm works in a bottom-up fashion in parallel with the recognition phase. Chiang and Fu simulated their extension. They pointed out its usefulness for pattern recognition applications.

2.3.6 The Chang, Ibarra and Palis array algorithm

Chang, Ibarra and Palis [Chang 87] designed a most interesting systolic algorithm (CIP) for parsing which is also based on CYK. Their algorithm works on an array of fixed size processors interconnected by one-way communication links. The algorithm first computes the recognition matrix and then traces the computation backward to recover the parse. The original version of the algorithm consists of

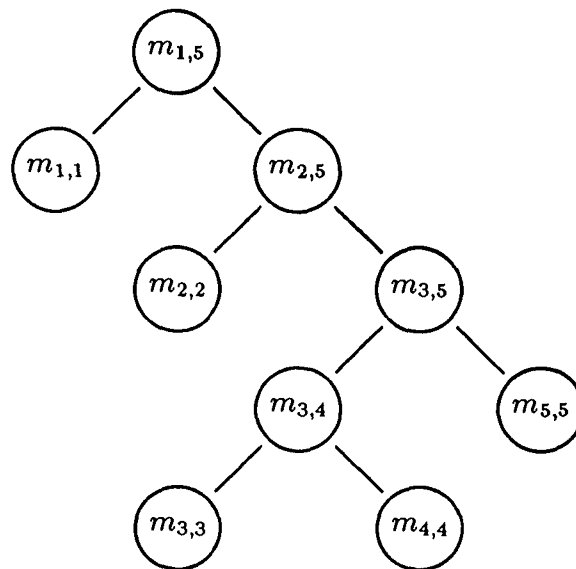


Figure 2–5: The parse tree for the CIP algorithm example.

seven phases. Two of these serve to reflect a matrix, horizontally first and then vertically. These inversions are necessary because of the constraint of one-way links between processors. To make this presentation simpler, I will assume the availability of an array of two-way interconnected processors. To assist the reader, I reproduce the example provided by Chang et al. in their paper and refer to it throughout the explanation. The example involves a five token input. The parse tree of the input is depicted in figure 2–5. A symbol $m_{i,j}$ in a node of this tree indicates that the non-terminal (or rule) for this node has been found in element $M_{i,j}$ of the recognition matrix. Which particular non-terminal (or rule) is not important. To avoid getting tangled up with synchronisation details when depicting data movements, I resort, as in the paper of Chang et al., to a technique of *unrolling in time and space* the content of the registers in the processors of an array. (What Chang et al. actually resort to is a special sequential multi-tape Turing machine model. They explain their algorithm with respect to this model and rely on a mapping [Ibarra 86] from algorithms for this model to algorithms for one-way rectangular systolic arrays to prove the existence of the parsing array. They have not produced, in their paper,

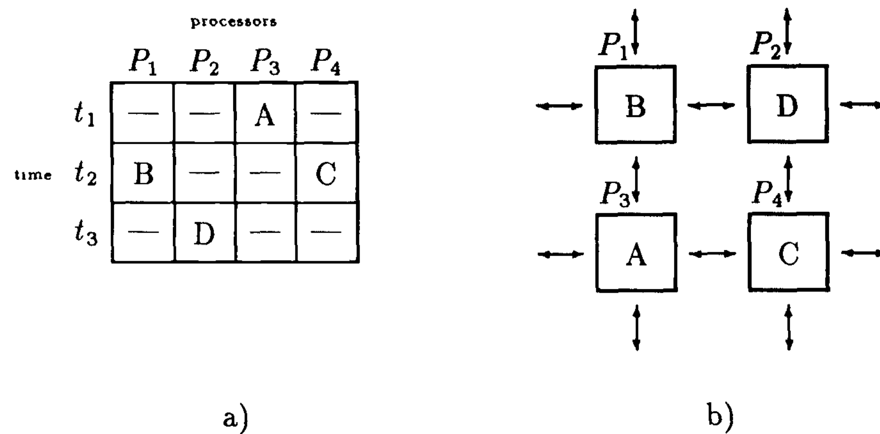


Figure 2–6: Two equivalent representations of processor contents: a) tabular form; b) *unrolled* form.

the systolic algorithm as such.) This technique consists of displaying in a processor the register contents that appear in this processor during the algorithm execution one beat after the register contents displayed in the processor to its left (or below) appear there (figure 2–6). As a consequence, the register contents displayed in the processors along a north-east/south-west diagonal all appear in these processors at the same time while those displayed in the processors along a south-east/north-west diagonal appear in them one after the other at two beats intervals. This technique allows one to scan a row of processors in a figure as if one was scanning it through time. When I explain the parse extraction phase, I will reverse the directions of *unrolling*.

The recognition phase

Like K-GKT, CIP computes the recognition matrix of CYK. Unlike K-GKT however, it does not establish a one-to-one correspondence between the processors of the array and the elements of the matrix. Furthermore, K-GKT requires that the whole input be available at the start of the execution (the algorithm is offline) while for CIP, the input is fed serially to the array during the execution (online).

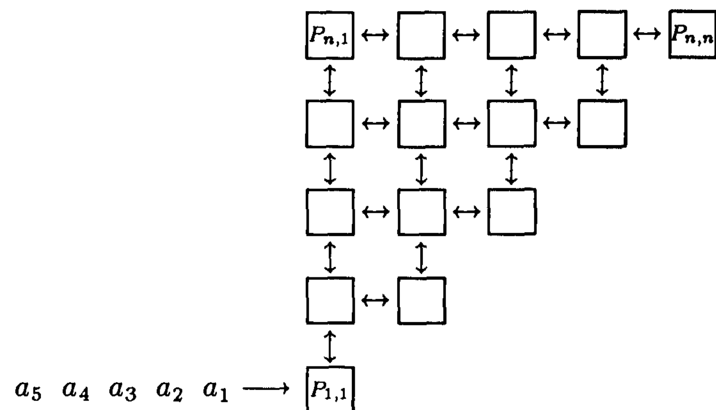


Figure 2–7: The array for the recognition phase of CIP.

The recognition phase of CIP uses an upper triangular array of processors as depicted in figure 2–7. I number the rows of this array from bottom to top and the columns, in the usual manner, from left to right. A token of the input is fed to processor $P_{1,1}$ on each beat. The computation of an element $M_{i,j}$ of the recognition matrix is distributed among the processors of row $j-i+1$. It starts on beat $2j-i$ in the first processor of the row and finishes on beat $3j-2i$ with the value of $M_{i,j}$ ending up in the last processor. I sketch out how the computation is carried out. The processors have four registers (among others) to hold the values of two pairs of matrix elements. On beat $2j-i$, processor $P_{j-i+1,1}$ generates an empty set (of non-terminals). On each beat, this set is passed from one processor to the next along the row and reaches processor $P_{j-i+1,j-i+1}$ on beat $3j-2i$. Each time a processor on the row receives the set, it adds to it the non-terminals obtained by the CYK combination over the two pairs of sets it holds at that time in its store. These last sets have been computed previously and have been passed to the processor by its left and bottom neighbours. The algorithm arranges for these sets to be just those needed for the computation of the set that is travelling from left to right, i.e. for $M_{i,j}$. Figure 2–8 shows the (unrolled) contents of the processors' four registers just mentioned during the recognition phase. Because this is unrolled, for each array,

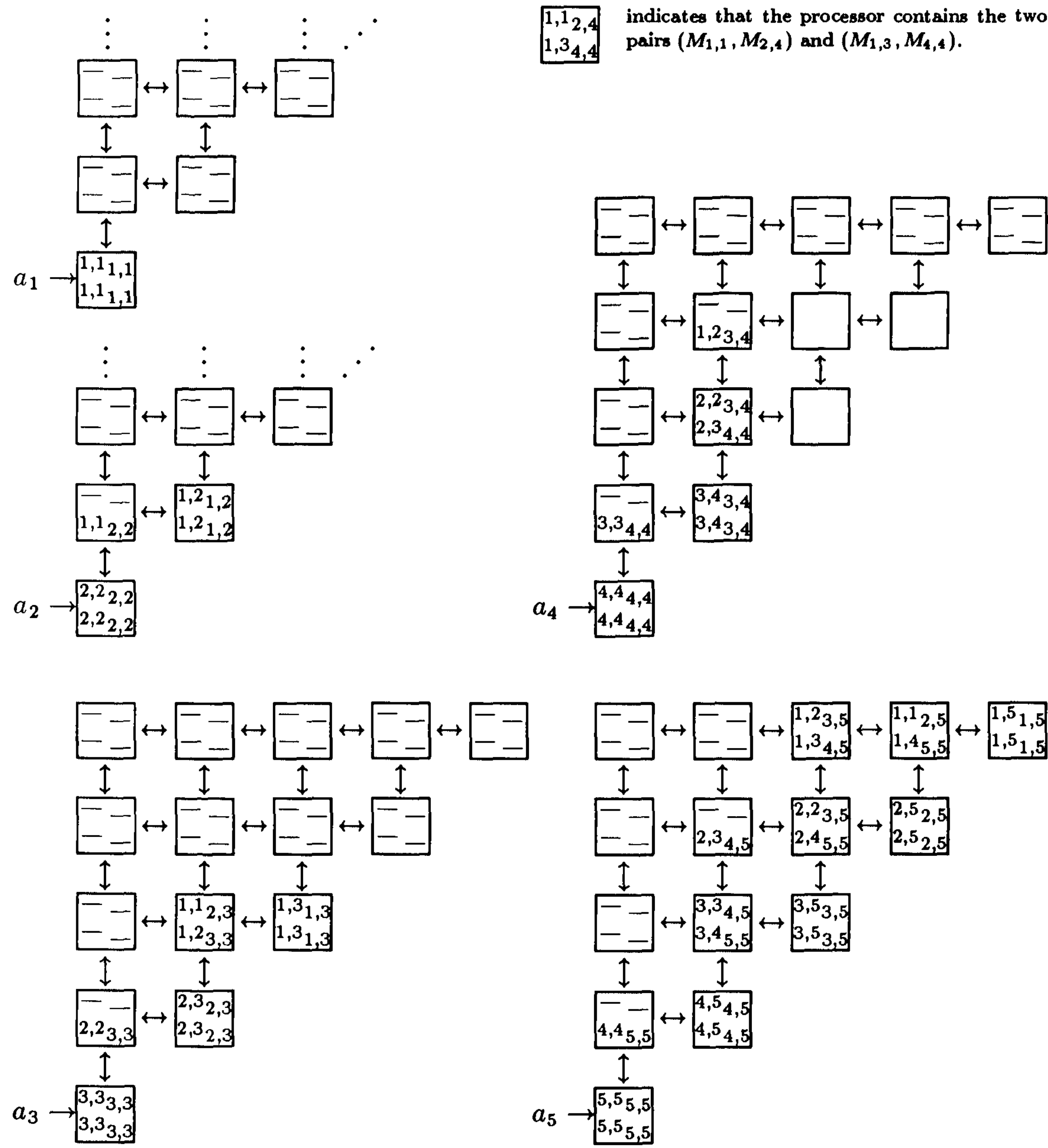
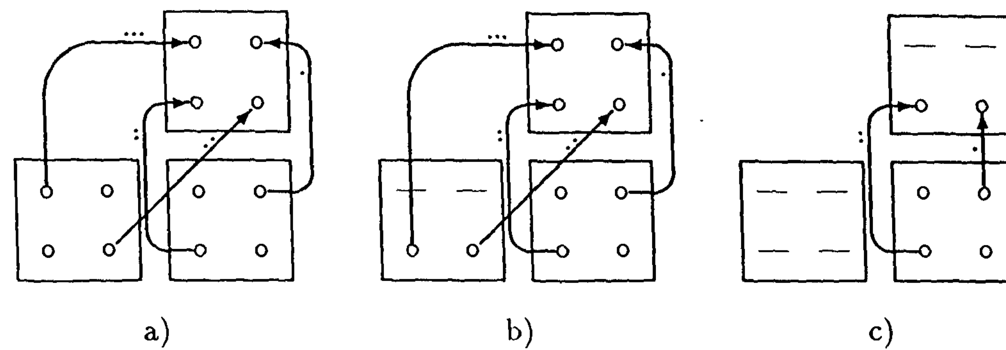


Figure 2–8: The contents of the processors during the recognition phase.



- A — depicts a null value while a o depicts a non-null one.
- The number of dots along an arrow indicates that the value travels from the tail to the head of the arrow in so many beats.
- The diagrams above cover three of five possible cases. Each case is determined by the number of null value pairs in the bottom processors. In the other two cases, the upper right processor ends up with four null values.

Figure 2–9: The data exchange occuring between processors during recognition.

what is displayed in processor $P_{5,5}$ appears in that processor 8 beats after what is displayed in processor $P_{1,1}$ appears in processor $P_{1,1}$. Note how the processors on the rows contain the matrix elements needed to compute the elements appearing on the rightmost processors of the rows. Figure 2–9 shows how the data are transferred from processor to processor to meet the algorithm's requirements.

Examining figure 2–8 and 2–9, we can make three observations. First, in the limit, half of the processors in the triangular array hold at all times only null values. We could easily adapt the algorithm to do away with these processors. Second, once a register has been fed its first non-null value, it only receives non-null values thereafter. Third, processors sometime need to pass to other processors values they have received two beats earlier. This implies that as with K-GKT, the processors need pipelines of two registers.

The parse extraction phase

To provide for parse extraction, during the recognition phase, processors record for each non-terminal in a set, the rule that cause the insertion. (Actually, only the rules themselves need to be recorded since the left-hand sides of the rules consist of just those non-terminals we are interested in.) If the grammar is ambiguous and more than one rule allows the insertion of a non-terminal, one is chosen arbitrarily. During the recognition phase, processors also record in separate registers copies of the first non-null values they receive in their working registers.

The phase has four sub-phases, two of which are executed simultaneously. The first sub-phase reconstitutes in reverse the data movements that occurred during the recognition phase. This backward trace of the recognition serves to spot the rules in the recognition matrix that are part of the parse. The second sub-phase has these rules migrate to another part of the array where they are *laid-out* in a representation of the parse tree. The third sub-phase flattens this tree and the fourth one outputs the result. Let us go through all this again more slowly.

Spotting the rules Recall how an element of the recognition matrix gets computed. An empty set is generated on the left boundary of the triangular array and as the set travels towards the diagonal, it encounters pairs of already computed matrix elements and gets augmented by the results of the CYK combinations over these pairs. (I assume from now on that the matrix elements are sets of rules rather then sets of non-terminals.) In the reverse process, a selected rule of the set will travel from right to left and while doing so it will encounter the same pairs of matrix elements the set met on the outward journey. Element $M_{1,n}$ must have a rule with the start symbol as its left-hand side. (If this is not the case, recognition failed and we have no parse to output.) This rule is the first to engage on its backward journey. It is the root of the parse tree. As it travels to the left, it is bound to meet

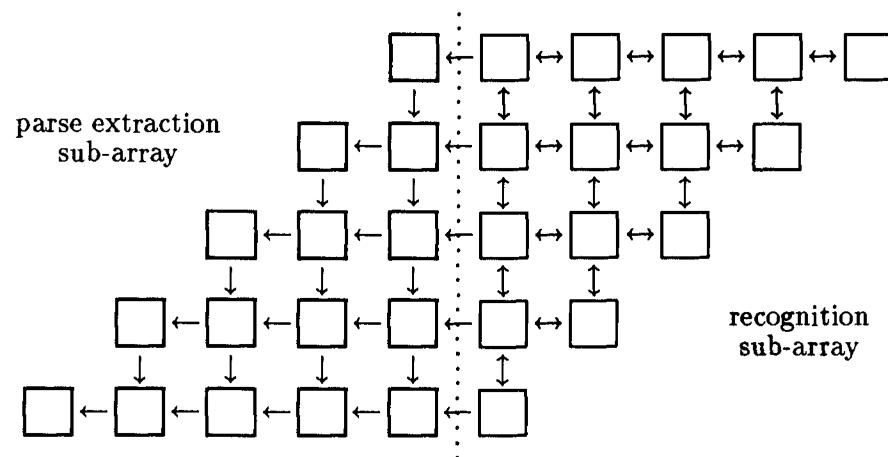


Figure 2–10: The parse extraction sub-array.

on the way the pair of sets containing the two rules responsible for its insertion in $M_{1,n}$. These two rules are the sons of the root rule and they are marked as being part of the tree. As the backward trace continue, the two sets containing the marked rules will eventually make their way to processors on the diagonal. From there, the rules will trigger the marking of the remaining rules of the parse tree, each, independently triggering the marking of the rules of its own subtree. When a set arriving in a processor of the diagonal contains no marked rule, a *null rule* is sent on a journey leftward along the row. I refer the reader to the paper of Chang et al. for further details. I simply mention here that the recording, during the recognition phase, of the first non-null values arriving in the processors' registers is a necessary step for the trace. These values play a role similar to the one that was intended for the pieces of bread Hansel left behind him as he walked through the forest with Gretel.

Laying out the rules To the left of the triangular array used for recognition is abutted another triangular array which is the vertical symmetric of the first turned upside down (figure 2–10). At the start of the parse extraction phase, each processor of this second array represents an empty *slot.* The rules of the first sub-

phase travelling from the diagonal of the first array towards its left boundary cross over in the second array and keep on migrating eastward until they find a vacant slot to occupy. At the end of this sub-phase, the non-null rules in the left array are laid out in a representation of the parse tree sought. A node is represented by a non-null rule in a slot, its right son by the nearest non-null rule below and its left son by the nearest non-null rule in the south-east direction. Figure 2–11 depicts the beginning of the backward trace and rule layout phases of the example. In this figure, I have reversed the directions of the unrolling. Hence, a value displayed in a processor appears there a beat after the values displayed in the processor above and in the processor to the right appear in these. Figure 2–12a depicts the left sub-array at the end of these two phases. Compare the representation of the tree in this figure with the tree of figure 2–5.

Flattening the tree and outputting the parse Through a *shift and accumulate* phase, a flattened representation of the tree is obtained in the bottom row of the array. The processors of the bottom row can hold two rules. The rules all migrate downwards. When a bottom processor receives a rule from its top neighbour while two rules already occupy its store, it sends one of these two rules, the one that arrived first, to its left neighbour. This flattened representation of the tree constitutes a rightmost parse of the input. Figure 2–12b shows the content of the left sub-array at the end of the shift and accumulate phase in the example. In a last phase, this parse is output through the leftmost or the rightmost processor at the base of the second sub-array.

Complexity

The time complexity of the algorithm is linear. The two-way links version presented here requires, more precisely, $7n$ beats to produce a parse of a string of length n. The

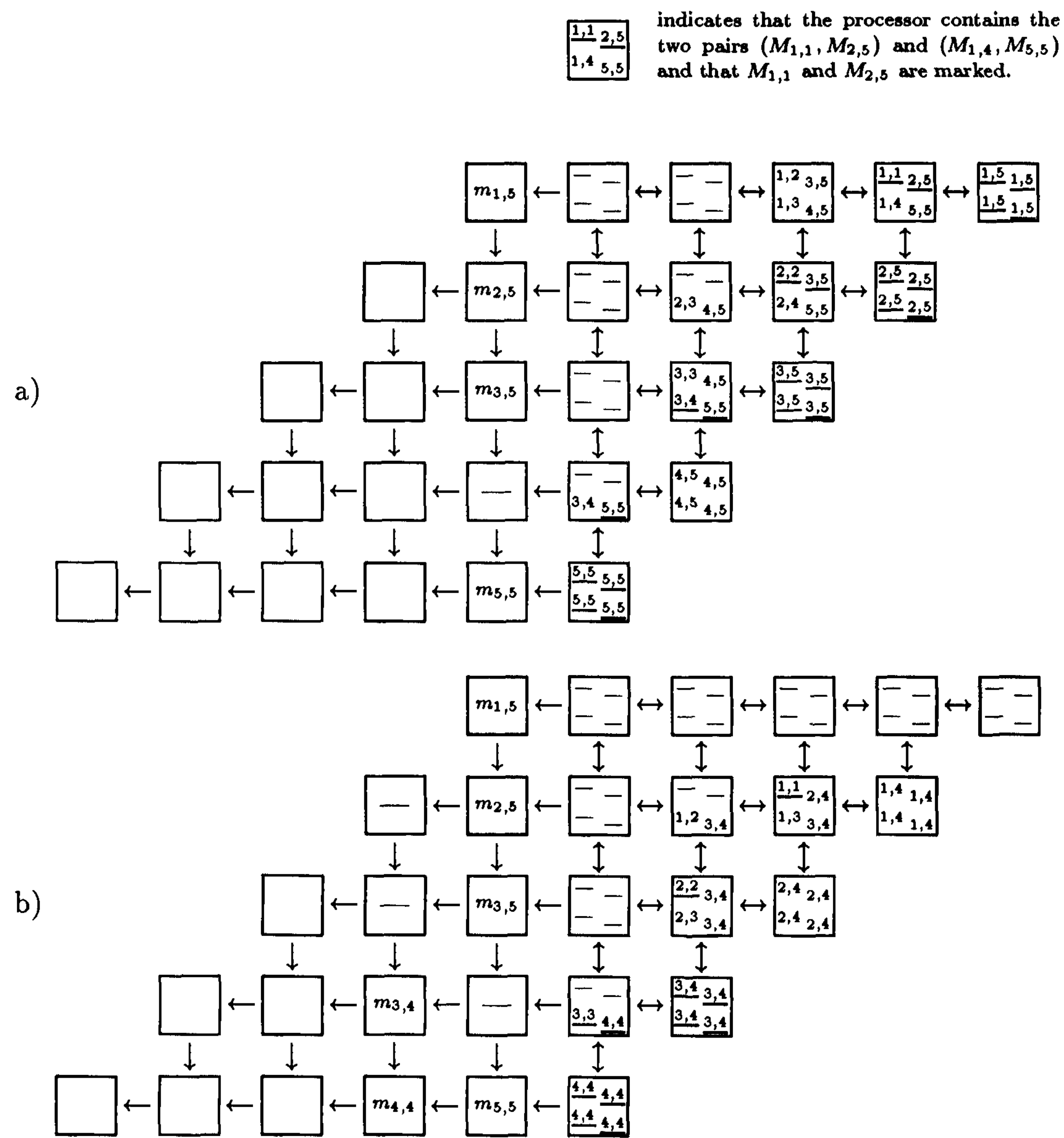


Figure 2–11: The contents of the processors during the beginning of the backward trace and rule laying out phase.

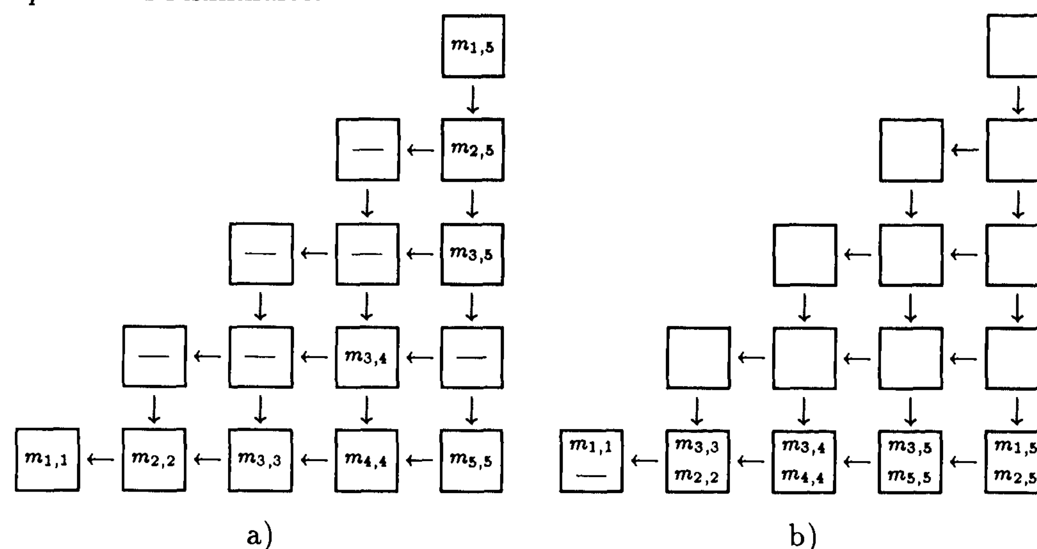


Figure 2–12: The content of the parse extraction sub-array: a) at the end of the rule layout phase; b) at the end of the flattening phase.

recognition phase and its backward trace require $2n$ beats each. The completion of the laying out phase, the tree flattening phase and the output phase each takes another n beats. The size of the processors of the array is completely independent of the input size. Thus, the space complexity is determined by the number of processors and is hence $O(n^2)$. The algorithm can cope with ambiguous grammar. However, as it stands, it cannot output all the parses of an ambiguous sentence since during the recognition phase, the array keeps a record of only the first reduction that is involved in the insertion of a given non-terminal in a given set.

2.3.7 Other work

Hirikawa [Hirakawa 83] wrote a PROLOG implementation of Earley's algorithm (the Chart parsing method). He targeted his program for execution on a concurrent version of PROLOG.

Fanty proposed the use of *connectionist networks* (neural networks) for parsing. Connectionist networks are based on a simplified model of the brain. They consist

of large collections of units with minimal computing capability of different kinds interconnected in a very irregular fashion. Typically, a unit will have many inputs and one output. Depending on its input, the unit can be in a *firing*[7] state in which case it sends a positive signal through its output line.

Fanty presented a scheme for building, from a grammar G, a connectionist network for parsing the strings of $L(G)$ of a fixed length n. Such a network is composed of $O(n^3)$ nodes consisting of one or two gates. The input is fed to the network by setting the relevant terminal nodes of the network into their firing state. Then, in a first phase, signals propagate from the terminal nodes to a single node corresponding to the start symbol and the whole sentence. If this node ends in a firing state, the input is accepted. In a second phase, signals propagate in the other direction and they set *on* the nodes corresponding to the nodes of the parse tree of the input. The whole process is extremely fast, it takes Cn time where C is of the same order of magnitude as the switching speed of the network's gates. The interconnection complexity of the network however is huge. Fanty has made no suggestion as to how the parse tree could be extracted from the network.

[7]This term is borrowed from neurology. A neuron, when stimulated with the right signals on its dendrites (inputs) sends a signal on its axon (output). It is then said to *fire*.

Chapter 3

An Extension to K-GKT

3.1 Introduction

As mentioned in the last chapter, the K-GKT algorithm implements on an array of processors the recognition phase of CYK. I propose a major extension to this algorithm for the implementation of the parse extraction phase.

Let us refer back to figure 2–3 (page 30) depicting the K-GKT array after it has processed the string (a+a)*a. Because E is a member of the set of processor $P_{1,n}$, we know that the input string is in the language generated by G_1. E was inserted in processor $P_{1,7}$ because of the T in processor $P_{1,5}$ and the $_*F$ in processor $P_{6,7}$. The T in processor $P_{1,5}$ was itself inserted there because of the $\underline{(}$ in processor $P_{1,1}$ and the E in processor $P_{2,5}$. Figure 3–1 expresses the situation succinctly. In this figure, lines join those processors whose values were involved in the insertion of the distinguished symbol in processor $P_{1,7}$ and lines join the processors on the diagonal to their corresponding tokens (only the relevant non-terminals are displayed). It is no coincidence that the schema obtained, ignoring non-involved processors, looks

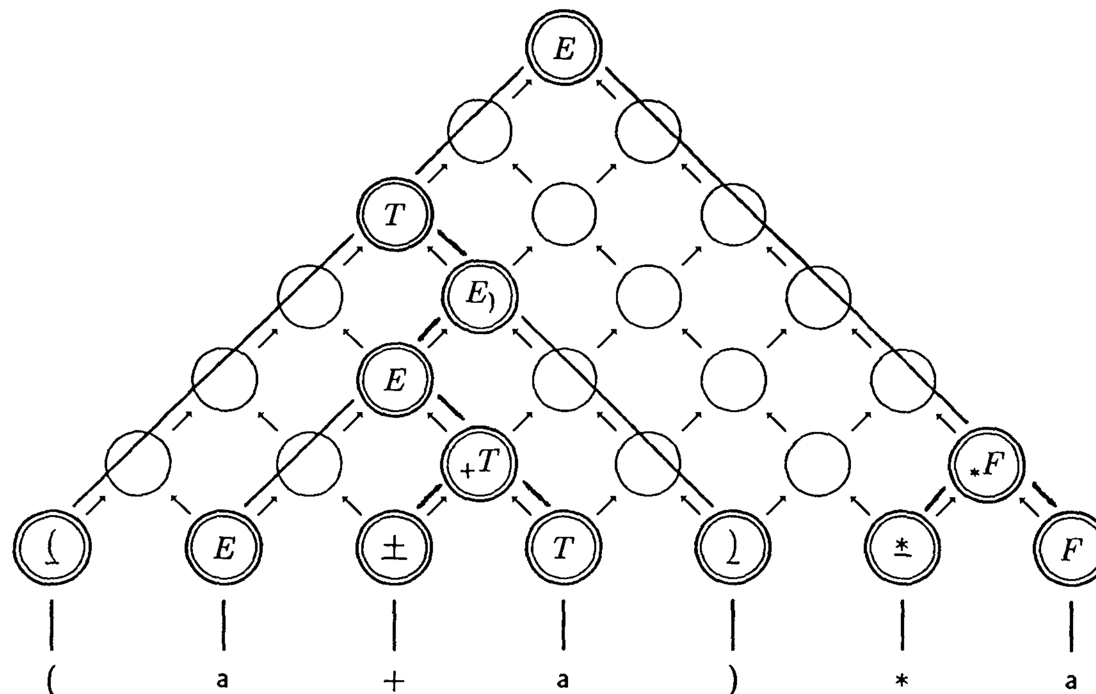


Figure 3–1: The underlying parse tree.

very much like a parse tree, it is one. The processors involved constitute the inside vertices of the tree, the lines constitute its edges and the tokens constitute its leaves. The reader can now understand why the array has been tilted in figure 2–3, and why it is tilted in figure 3–1. In the remaining of the dissertation, I shall continue to depict arrays of processors in this fashion. Notice that I number the processors as if the array was not tilted. Hence, I denote the leftmost processor at the base of the pyramid by $P_{1,1}$, I denote the rightmost one by $P_{n,n}$ and I denote the processor at the top by $P_{1,n}$. I shall also refer to a line of processors on a forward diagonal of the tilted array as a "row" of processors (as if the array was not tilted) and I shall refer to a line of processors on a backward diagonal as a "column" of processors.

The proposed modification to the K-GKT algorithm involves a reconfiguration of the array to give the conceptual lines joining the processors on figure 3–1 a physical reality. The basic idea is to use the processors between the vertices of the underlying tree as communication links, that is, to use them for passing information.

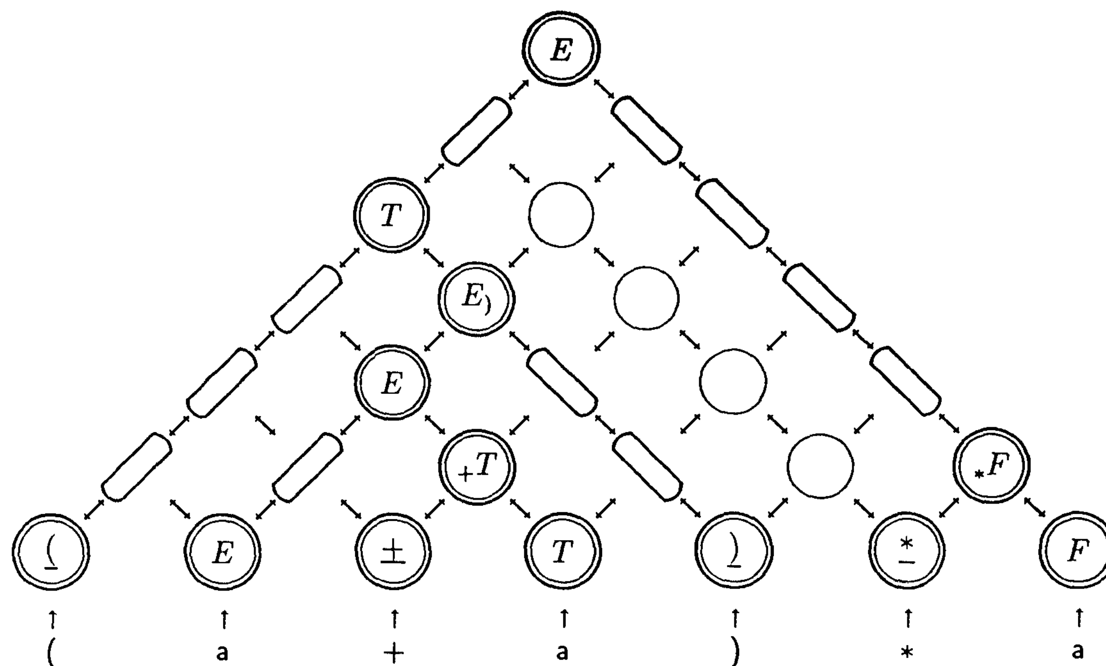


Figure 3–2: The array after reconfiguration (marking).

The extended algorithm has three phases. I call them the *recognition* phase, the *marking* phase and the *output* phase. The recognition phase corresponds almost exactly to the K-GKT algorithm, the marking phase reconfigures the array and the output phase outputs the parse. The marking and the output phases proceed only if recognition is successful (only on valid inputs). The heart of the extension consists of the addition of two counters to each processor. At all time when a processor is computing its value during the recognition phase, these counters indicate where the information currently on the processor's fast and slow belts originated from. Processors save the values of their counters whenever the information on their belt causes (via the CYK combination) an insertion of a non-terminal in their set. The marking phase employs the stored counter values to locate relevant processors and *mark* them as either *tree node* or *link node* processors. The former are the internal vertices of the underlying tree while the latter are those in-between. Figure 3-2 shows the array of our example after reconfiguration (see key in figure 3-5). Finally,

the output phase uses the reconfigured array to output a parse of the input string. Recall that a parse of a string is a sequence of (numbers of) rules that we can apply to derive the input string from the distinguished symbol. The third phase outputs these rules sequentially via processor $P_{1,n}$.

In this chapter, I consider the case where we are interested only in obtaining one parse of the input string. If the grammar is unambiguous, a valid string will only have one parse but if it is ambiguous, a string may have many parses. In the latter case, if we require only one parse, we "choose" it arbitrarily. Chapter 5 deals with the case where we want to obtain all the parses of an input string.

3.1.1 Definitions

Before going into the details of the three phases of my algorithm, I define various terms to simplify the presentation. An array of n rows (or columns) is said to be of *size n*. A processor not on a boundary of the array has four neighbours. In the general context, I call them the *upper left*, *upper right*, *lower left* and *lower right* neighbours. For a marked node, only one of its two upper neighbours will be relevant. I will call this neighbour simply the *upper* neighbour. Likewise, I will refer to the relevant lower neighbour of a link node as its *lower* neighbour. I also employ some of the usual graph theory terminology and speak of the *sons* and/or the *father* of a tree node[1]. I call the son of a tree node and the processors (if any) leading to the son a *branch* of the father node. This is different from a *branch* in graph theory. Node $P_{1,n}$ will always constitute the root of the underlying tree so I often refer to this node as the *root* of the tree. By extension, I also refer to it as the root of the array. The root is said to be on *level* 1, its two lower neighbours

[1]The father or the son of a tree node will itself always be another tree node.

are said to be on *level* 2 and so on. An array of size n has n levels. I call level n the *base* of the array. (The base in the tilted array is what I referred to it as the *diagonal* in the non-tilted array.) A processor at level k is said to be at *distance* $|k - l|$ from a processor at level l. The *height* of an array is the distance between its root and its base ($height = size - 1$). Recall that a processor $P_{i,j}$ contains the set of non-terminals deriving the portion of the input string starting at position i and ending at position j. I call this sub-string the string *spanned* by the processor.

3.1.2 Desiderata for systolic algorithms

Kung [Kung 79] has listed the following properties as desirable for a systolic algorithm:

1. that all processor operations take the same time.

2. that each processor requires a fixed amount, as small as possible, of storage.

3. that the processors be identical except maybe for special cases such as processors at the boundary of the array.

4. that the communication geometry be simple and regular.

5. that the data movement be simple and regular.

These properties ensure that the algorithm may be easily and efficiently implementable in VLSI systems. It is simple to produce an integrated circuit composed of a large collection of identical components interconnected in a regular fashion. If the processors are all the same, then building a bigger array simply involves adding more processors. The algorithm that I propose satisfies all of these properties except for 2. As we will see later, for an $n \times n$ array, the storage requirement of each

processor is proportional to $\log n$. As a consequence, processors built for an array of some given size may not be usable for arrays of bigger sizes. I shall argue, in the last chapter, that this drawback may not be significant in practice.

3.2 The algorithm

3.2.1 The recognition phase (K-GKT plus counters and pointers)

Each processor contains two counters. While a processor is computing its value, the algorithm arranges for these counters to hold the values of the distances in between this processor and the processors which computed the values currently on the processor's fast and slow belts. When a pair of processor values causes an insertion of a non-terminal in the set of the processor, the counters' values are saved. These saved counter values will be used by the marking phase. We can think of them as *pointers* to the processors responsible for the insertion. If our grammar is ambiguous, more than one processor value pair may cause insertions. For the moment we are interested in obtaining only one parse of the input string so we assume the processors record only one pointer pair per non-terminal.

The algorithm arranges for the counters to hold the right values at the right time as follows. A processor at a distance d from the base computes its value from time $t = \lfloor 3d/2 \rfloor$ to time $t = 2d - 1$ [Kosaraju 75]. During this interval, at time t, the two values on the processor's slow belts are from the processors below (on either the same row or column) that are at a distance $t - d + 1$ away while those on the fast belts are from the processors at a distance $2d - t$ away. For the counters to take the values of these distances at the right times, they simply need to be initialised at the values $\lfloor d/2 \rfloor + 1$ and $\lceil d/2 \rceil$ at time $t = \lfloor 3d/2 \rfloor$ and to be

EX BIBL. UNIV. EDINBURGH

respectively decreased and increased by 1 at each time unit afterwards. Various methods can be used to achieve the initialisation. I propose two, both based on control signals. The first assumes that at the start, the counters are preset, one at value zero, the other at value one. The second considers on the contrary that the counter values are initially undefined. Figures 3–3 and 3–4 show how each method performs the initialisations.

With preset counters

The method resorts to two sets of control signals that are sent from the base at time zero. The signals of one set must all travel either along the rows of the array or along its columns. From the start until they receive a first control signal (depicted in figure 3–3 by ↑), the processors increment one of their counters, the lower one, at every odd beat and the other at every even beat. The signals of the first set visit a new processor on every beat. A processor at distance d from the base will thus receive a signal from this set on beat d. At that time, one of the processor's counters will hold the value $\lceil d/2 \rceil$ while the other one will hold the value $\lfloor d/2 \rfloor + 1$. From then until a second control signal is received (depicted in figure 3–3 by ↕), the processors keep their counters' values intact. The signals of the second set travel at the speed of 2/3 processor per beat. (An extra set is not really needed here. We can use the signals from GKT for the fast to slow belt transfer.) They reach processors at distance d on beat $\lfloor 3d/2 \rfloor$. After it has received the second control signal, a processor decrements its lower value counter and increments the other one at every beat.

With counters undefined

This method also resorts to two sets of control signals. As in the previous method, the signals of the first set (depicted in figure 3–4 by ⇑) travel at the speed of

Processors

Time	1	2	3	4	5	6	7
0	↕↑ 0 1	0 1	0 1	0 1	0 1	0 1	0 1
1		↕↑ 1 1	1 1	1 1	1 1	1 1	1 1
2		↕	↑ 1 2	1 2	1 2	1 2	1 2
3			↕ 1 2	↑ 2 2	2 2	2 2	2 2
4				↕ 2 2	↑ 2 3	2 3	2 3
5				↕ 1 3	2 3	↑ 3 3	3 3
6					↕ 2 3	3 3	↑ 3 4
7					1 4	↕ 3 3	3 4
8						↕ 2 4	3 4
9						1 5	↕ 3 4
8							2 5

Figure 3–3: Counter initialisation with preset counters.

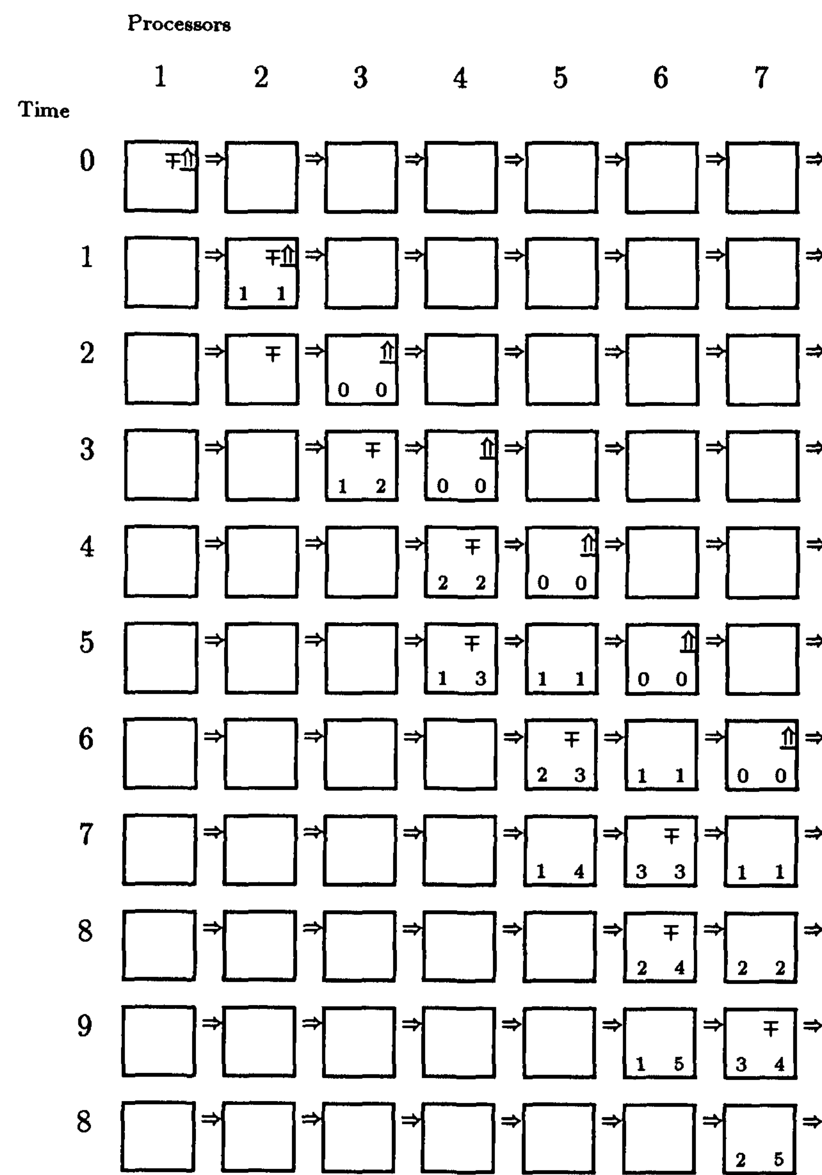


Figure 3–4: Counter initialisation with undefined counters.

1 processor per beat and those of the second set (depicted in figure 3–4 by $\mp$) travel at the speed of 2/3 processor per beat. The signals of the second set have two states. I call them the *even* and the *odd* states. The signals leave the processors of the base in an even state and they toggle their state whenever they arrive at a new processor. The state indicates if the processor the signal is currently visiting is at an even or an odd distance from the base. The 2/3 processor per beat speed is achieved by keeping the signals one beat in each processor and an extra beat in alternate processors. For the method to work, we must have the two beat pauses occur in odd distance processors. This ensures that the signals arrive in processors at distance d at time $\lfloor 3d/2 \rfloor$. At the start of the array operation and until they receive the first control signal, the processors leave their counters undefined. When they receive the first signal (on beat d for processors at distance d) the processors initialise their two counters to zero. On each subsequent beat up to and including the beat when they receive the second signal, they increment both counters by one. The second signals arrive $\lfloor d/2 \rfloor$ beats after the first signals at processors at distance d and hence, the operations above bring the values of these processors' counters to $\lfloor d/2 \rfloor$. (Note that these operations involve no increment operation for processors at distance 1 which receive both signals at the same time.) The processors then increment either one or both of their counters by one, depending on whether the second signals are in an even or an odd state. This final operation brings the values of the counters to $\lfloor d/2 \rfloor + 1$ and $\lceil d/2 \rceil$ on beat $\lceil 2d/3 \rceil$ as required.

3.2.2 The marking phase

Once recognition is complete, if the input string is accepted, the array goes into the marking phase. This phase marks the processors representing the internal vertices of the underlying parse tree as *tree nodes* and marks those in between as *link nodes.* The root initiates the marking. The distances from the root to its sons are indicated

by two pointers (counter values) saved by the root during recognition. The root uses these pointers to mark its two branches. This involves marking its sons as tree nodes and the processors in between it and the sons as link nodes. Once marked, the sons initiate the marking of their own branches and so on until the base nodes get marked (as tree nodes).

I suggest two methods for the marking of the *branches* of a node. One entails the passing down of pointers while the other involves only token passing.

Marking by *pointer passing*

The father decrements both pointers by one and sends each decremented value to the relevant lower neighbour. A processor, upon receiving a pointer, if the pointer has value zero, marks itself as a tree node (and initiates the marking of its sub-tree), otherwise, it marks itself as a link node, decrements the pointer by one and passes it on the next beat to its lower neighbour, i.e. the one opposite the upper neighbour that sent it the pointer. One can easily convince oneself that this method will mark a branch of d processors in d beats. I do not provide a proof.

Marking by *token passing*

On each beat, the father node sends to each of its lower neighbours a token and decrements its two pointers by one. The father stops sending tokens to a neighbour when the pointer associated with it reaches zero. When a processor receives a token for the first time, it keeps it, i.e. it records the event. If on the next beat it obtains another token it marks itself as a link node and on the following beat, it passes the second token received to its lower neighbour. Afterwards, it keeps on passing the tokens it receives on one beat down to its lower neighbour on the next beat. If after getting its first token, a processor does not obtain another one on the following

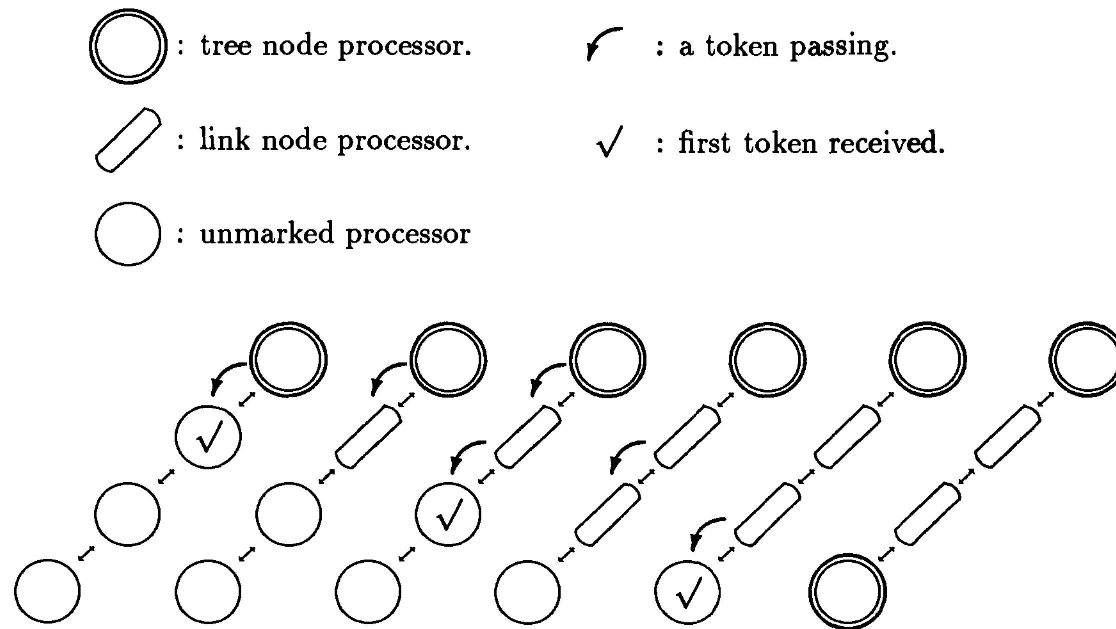


Figure 3–5: The marking of a branch by token passing.

beat, it marks itself as a tree node. Figure 3–5 depicts the marking of a branch of length 3.

It is not immediately obvious that this method will do the job. I informally indicate here why it works. Later, in the section on complexity analysis (section 3.3.2, page 64), I provide a more formal proof. Suppose a tree node is at a distance d from its son. The father's pointer associated with this son holds the value d and it will send d tokens down the branch leading to the son. The first processor on this branch receives d tokens, keeps one and transmits the $d-1$ others. The second processor receives these $d-1$ and transmits $d-2$ and so on. The dth processor, the son, receives only one token and marks itself as a tree node while every processor between the father and the son has received at least two and marked itself as a link node.

Passing down more information

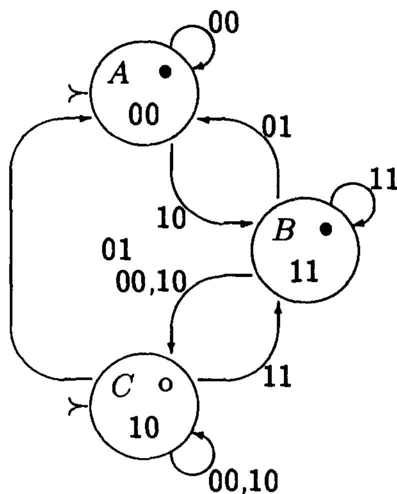
In the case we consider in this chapter, processors never need to record more than one pointer pair per non-terminal. As I will show in section 3.3.1, they may need to record only one pair or as many pairs as there are non-terminals in the grammar, depending on the type of grammar used. If they hold at most one pair, processors marked as tree nodes initiate the marking of their branches using the unique pair they hold. If processors hold many pairs, processors marked as tree nodes will need to select one pair amongst those. Each pointer pair will be associated with a different non-terminal and a different rule. A father can thus inform its son which pair to use by indicating to it which non-terminal is relevant. This information can be passed down with either the pointers or the tokens (depending on the method chosen) used in the marking phase. The rule associated with the pair which the father uses indicates which terminals are relevant to its sons. The root's relevant non-terminal will always be the grammar's starting symbol.

3.2.3 The output phase

After the marking, the array goes through a phase during which it outputs the parse of the input string. I assume that during the recognition phase the processors recorded the rule numbers responsible for the insertion of non-terminals. I show how the string of rule numbers composing the parse can be output sequentially at the root. I explain this for a parse output in leftmost order. (I indicate later how we can adapt the method for other parse orders.) The output can be likened to a *structured bucket brigade*. The volunteers are the processors and the buckets are the rule numbers. All the rule numbers must be passed to the root which outputs them. The root first outputs its own rule number. Then it outputs the rule numbers it receives from its lower left neighbour. The last rule number sent by this neighbour is accompanied by an *end-of-parse* marker. The root strips the rule

number of this marker before transmitting it. It then transmits the rule numbers sent by its lower right neighbour. The last rule number from the right is also marked. The root transmits this one with the mark however since it constitutes the last rule number of the whole parse. During the phase, link nodes simply pass to their upper neighbour the rule numbers they receive from their lower neighbour. Tree nodes act exactly as the root except that instead of outputting values they send values up to their upper neighbour. The base nodes, which are tree nodes, constitute a special case. They have only one value to send up, namely, their own. This is the last value of their own sub-parse so they attach an *end-of-parse* marker to it.

To satisfy desideratum 2 of section 3.1.2, I impose the following constraint on processors: they can hold at most one rule number at any given time (not counting the tree nodes' own rule number and rule numbers that may be held in auxiliary registers for data transfer). As a consequence of this constraint, during the output, upper neighbours of nodes may be unable to accept a rule number because they already hold one that they themselves cannot send up. The bucket brigade is then held up at these points. (This will always be due to ancestors delaying the output of their right sub-parse while they output their left sub-parse. The right sub-parse is blocked in that case in the right sub-tree.) To control the flow of rule numbers up the array, processors operate as follows. On alternate beats, they can either *send* or *receive* information. While the processors on even numbered levels are on *sending* beats, those on odd numbered levels are on *receiving* beats and vice-versa. As just mentioned, processors have storage space to hold one value (rule number). They constantly try to obtain values from (either of) their lower neighbour(s) to fill this storage and simultaneously, they try to get rid of any value in their storage by sending such value to their upper neighbour. To obtain a value from a lower neighbour, a processor sends the lower neighbour *requests* for a value until the neighbour provides a value. Before sending a value up, a processor must first have



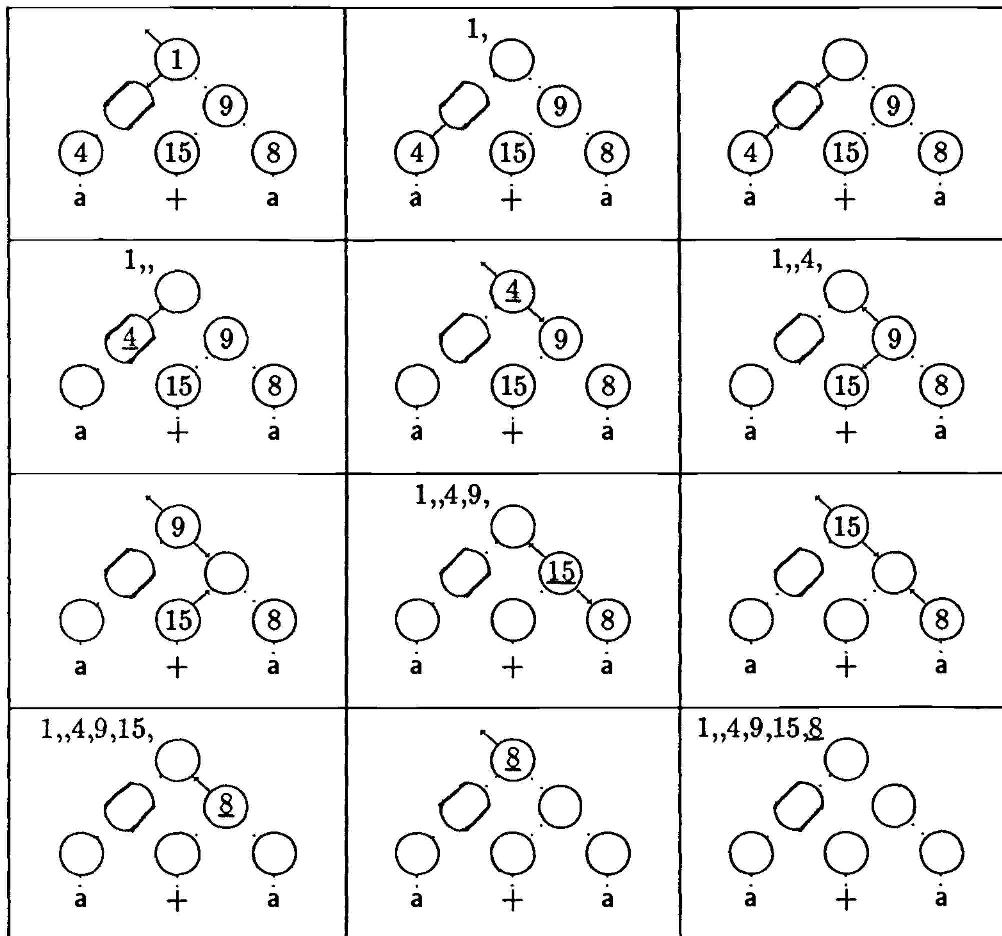
- A processor's next state depends on what it received on the previous (receiving) beat. The transition arcs are labeled by one or two pairs of digits. The first digit of the pair indicates whether the processor received a request (1) or not (0). The second indicates if it received a value.
- Using the same convention, the two digits in a state circle indicate what the processor will send on the subsequent (sending) beat.
- The circle in the top right of a state circle indicates whether the processor in this state holds a value to send up (•) or not (o).

Figure 3–6: State diagram for the output phase.

received a *request* from its upper neighbour. Figure 3–6 shows the state diagram the processors go through during the output phase. The diagram has two initial states, A and C. A processor starts in state A if it is a tree node and if the first operation to be executed is to send up its own rule number (it may not be the case, see below). A processor otherwise starts in state C. Figure 3–7 shows the output of the parse of the string a+a according to grammar G_1.

3.2.4 Parse orders

The output phase achieves the equivalent of a traversal of the parse tree during which we output the rule numbers of the nodes we visit. The three operations



- The twelve rectangles show the content of the array on successive beats during the output.
- The dotted lines between the processors and the processors and the token depict the edges of the underlying tree.
- The arrows indicate the non null data transfers occuring during the beats. Down arrows are for requests and up arrows are for values (rule numbers). The effects of the transfers are apparent on the rectangle depicting the content of the array on the following beat.
- Rule numbers with *end-of-parse* markers are underlined.
- A null value separates the first and the second rule number of the parse because on beat three, the root did not output a value.

Figure 3–7: The output of the parse of a+a.

Node operations	Tree Traversal Order	Parse Order
SOR-TLS-TRS	preorder	leftmost
SOR-TRS-TLS	inverse preorder	rightmost
TLS-SOR-TRS	infix order	—
TLS-TRS-SOR	postorder	inverse rightmost
TRS-SOR-TLS	inverse infix	—
TRS-TLS-SOR	inverse postorder	inverse leftmost

Table 3–1: Node operations, tree traversal and parse orders.

of the tree node processors: SEND OWN RULE (SOR), TRANSMIT LEFT SUB-PARSE (TLS) and TRANSMIT RIGHT SUB-PARSE (TRS) correspond to the familiar tree traversal operations: VISIT NODE, TRAVERSE LEFT SUB-TREE and TRAVERSE RIGHT SUB-TREE. In the description above, I chose to perform the operations in the order SOR-TLS-TRS and obtained a parse in leftmost order. I could however have chosen any other order. For example, had I wished the parse to be instead in reverse rightmost order (the order produced by the Shift-Reduce parsing methods), I would have opted for the order TLS-TRS-SOR. Table 3–1 lists the tree traversal and parse orders corresponding to the various orders of execution of the tree node operations.

3.3 Complexity

I now analyse the space and time complexity of my algorithm. In this analysis, I derive complexity measures solely as a function of the length of the input string. I do not consider the size of the grammar in spite of the fact that in some applications such as natural language, the grammar size can be a predominating factor.

3.3.1 Space

For an input string of n tokens, the array must consist of $n(n+1)/2$ processors. Were it not for the counters and pointers, the space requirements of each processor would be independent of the array size (i.e. constant). The processors have only two counters but they may have to save many pairs of pointers. The counters and pointers need to be of length at most $\log n$. This chapter deals only with unambiguous grammars. If a grammar is unambiguous, there exists at most one derivation from any non-terminal to a given string. Hence a processor will never need to hold more pairs of pointers than there are non-terminals in the grammar. In relation to the input size, this is a constant and so the space complexity of the whole array is $O(n^2 \log n)$.

3.3.2 Time

I now analyse the time complexity of the algorithm. I analyse each phase separately. We shall see that each phase takes linear time and thus that the whole algorithm takes linear time.

Recognition

As mentioned earlier, the CYK algorithm implemented on the K-GKT systolic array has a linear time complexity. The recognition phase of our algorithm differs from K-GKT only by its use of counters. The only operations our algorithm requires to perform on the counters are: SET TO ZERO, INCREMENT (by one), DECREMENT (by one), NO OPERATION and TEST IF ZERO. It is possible to implement the counters so that these operations can all be executed in constant time. In section 3.4, I present a detailed implementation of such constant time counters together with a

proof of its correctness. Assuming we implement the counters this way, we may conclude that the recognition phase has a linear time complexity.

Marking

The root initiates the marking phase and the phase is terminated when every processor of the underlying tree has been marked. I prove that the phase completes in linear time. Observe that n, the size of the input, is related to the height of the array, h, by the following relation: $h = n - 1$. It is thus sufficient to show that the phase terminates after a time proportional to h.

Theorem 3.3.1 *For an array of height h holding a valid underlying parse tree, $2h$ beats after the marking of the root, every processor of its underlying parse tree will be marked.*

Proof By induction on h. The conclusion is true for $h = 0$. The array then consists of the root only and once it has been marked, the marking is over. Let us assume that the hypothesis holds for any array of height less than some $h > 1$. I shall prove that it also holds for any array of height h. For an array of height h greater than zero, the root has two sons. The argument presented below applies to both sons so let's consider only one. Suppose the left son is at a distance d from the root. Then, the son is itself the root of a sub-tree of height $h - d$. By the induction hypothesis, $2(h - d)$ beats after the son has been marked, every processor of its sub-tree will be marked. What is left to prove is that the son and the processors linking it to the root will be marked within $2d$ beats after the root has been marked. I show this by induction on d. A son at distance 1 receives the token the root sends on the first beat. On the next beat, it receives none and marks itself as a tree node, so the conclusion is true for $d = 1$. Let's suppose the hypothesis is true for a son at distance less than some $d > 1$ and prove that it also holds for a son at distance

d. Consider the node in between the son and the root and nearest to the root. As far as the proof is concerned, after two beats this node will behave exactly like a root that would be at distance $d-1$ from the son. On the first beat, it will receive the first token sent by the root. On the second, it will receive the second token. From then on, it will pass down to its lower neighbour $d-1$ tokens just as a root at distance $d-1$ would. By the induction hypothesis, the son will thus be marked $2(d-1)$ beats later, i.e. after beat $2d$. It follows that $2h$ beats after the marking of the root, every processor of the underlying left sub-tree, and by the same argument, every processor of the underlying right sub-tree will be marked. ∎

Output

The root must output as many rule numbers as there are internal nodes in the parse tree. If we take out the leaves of the parse tree (i.e. the terminals) we are left with a full binary tree (one whose vertices all have either two sons or none) of n sons (the n processors at the base). This tree thus consists of $2n-1$ nodes and correspondingly, the parse consists of $2n-1$ rule numbers. If we can show that a linear time after the start of the output phase, the root outputs a rule number on each of its sending beats until the last value has been output, we will have shown that the output phase takes linear time. The next theorem serves that end.

Theorem 3.3.2 *For an array of height h holding a valid underlying parse tree, $2h+i$ beats after the start of the marking phase, every node of the underlying tree at level $n-i$ is ready to send values up on each successive sending beat until there are no more values left to send up.*

Proof By induction on i. The conclusion is true for $i=0$. After $2h$ beats, the nodes at the base are marked and they are ready to send up their own value. Suppose the statement is true for some $i-1 \geq 0$, let's prove that it is also true

for i. Consider a node at level $n - i$. Suppose it is a link node. It then has one lower neighbour at level $n - i + 1$. Let's call the node on level $n - i$ the *upper* node and the one on level $n - i + 1$ the *lower* node. By the induction hypothesis, after beat $2h + i - 1$, the lower node can send a value to the upper node on each of its subsequent sending beats. These sending beats correspond to the receiving beats of the upper node. Thus from beat $2h + i - 1$, the upper node will always be able to fill its storage with a value, if need be, by obtaining one from the lower node. (When a node sends a value up, it also sends a request down so as to obtain a new value on the following beat). Thus on each sending beat after beat $2h + i$, the upper node will be able to send values up. The same argument applies to the case when the node at level $n - i$ is a tree node. ∎

The root is never impeded from sending values up. So if it is ready to send a value, it sends it. We thus conclude from theorem 3.3.2 that after $3h$ beats ($2h + (n - 1) = 3h$), the root is outputting values at every sending beat. Since there is a linear number of values to output, it follows that the output phase has linear time complexity.

When the parse is output in reverse leftmost (respectively reverse rightmost) order, the first value to be output will be from either the leftmost (respectively rightmost) base node. Let's suppose, without loss of generality, that it will be from the leftmost one. This node will be marked after beat $2h$. It will take another h beats for its value to migrate to the root. Thus in this case, the first value of the parse will be output on beat $3h + 1$. The $2n - 2$ remaining values will be output during the next $2n-2$ sending beats. The whole phase thus requires exactly $(3h + 1) + 2(2n - 2) = 7n - 6$ beats. If the parse is output in either leftmost or rightmost order, at least 1 and up to $n - 1$ values might be output before the first base node value gets out on beat $3h + 1$. Thus in this case, it will take in between 2 and $2(n - 1)$ less beats to output the parse than in the previous case (i.e. in

between $5n - 4$ and $7n - 8$ beats). The exact number will depend on the number of tree nodes on either the upper-left or the upper-right boundary of the array.

I have shown that the execution of each of the three phases of the extension takes a linear time. Consequently, the whole algorithm execution takes a linear time.

3.4 Constant time counters

If we implement the counters in the usual way, because of carry and borrow propagation delays, the increment and decrement operations would each take a time proportional to $\log n$ where n is the biggest value the counters can hold. The only operations the algorithm needs to perform on the counters are: increment (INCR), decrement (DECR), no operation (NOP), set to zero (SET0) and test if zero (IF0). I show how by resorting to a *carry save* vector, a *borrow save* vector and another vector which I call the *significance* vector, we can implement the counters such that all of the above operations require constant times.

3.4.1 Implementation

A counter consists of four bit vectors: the *value* vector (V), the *carry* vector (C), the *borrow* vector (B) and the *significance* vector (S). The vectors are all of length $\log n$. The value vector holds the interim value of the counter while the carry and borrow vectors hold pending carries and borrows that have not yet fully propagated through the counter. The significance vector indicates the position of the most significant non-zero bit in the value or the carry vector. I refer to this position as the MSB position. If the carry and value vectors have only zero bits, the MSB position is position 0, the position of the least significant bit. The MSB position is

indicated by the least significant bit at one in the vector S. For the purpose of this explanation, I partition the counter in vertical *slices*. Each slice consists of the bits of the four vectors in a given position. If you want, you can see the counter as a vector of slices. The slice in position i consists of the four bits V_i, C_i, B_i and S_i. If $C_i = 1$ ($B_i = 1$), I say that slice i holds a carry (borrow). I refer to the slice in the MSB position as the MSB slice. I refer to the value of a given bit at some specific time t by superscripting its symbol with t. Hence, V_i^t refers to the value of the V bit in slice i at time t. Likewise, MSB^t refers to the MSB position at time t.

On each beat (of the systolic array) each slice of the counter changes the value of its bits according to a function of the current values of its own bits and of the bits of its two neighbouring slices. I refer to this function by the symbol $\mathcal{F}$. Table 3–2 contains a definition of $\mathcal{F}$ in truth table form. In table 3–2a, not all possible combinations of values for V_i^t, C_{i-1}^t, B_{i-1}^t and S_{i-1}^t are present. As will be shown later, those missing can never occur (see lemma 3.4.3). Slice 0 behaves as if there was a virtual slice -1 and changes the value of its bits using almost the same function as the other slices. This virtual slice has all its bits at 0 except when the operation INCR or DECR are performed. The INCR operation is performed by setting the C bit in slice -1 to 1 just before the slices update their value while the DECR operation involves setting the B bit to 1. The operation SET0 is performed by setting (V_0, C_0, B_0, S_0) to the value $(0, 0, 0, 1)$. The operation IF0 is performed by testing if slice 0 has this value. I refer to this last fact by saying that the counter is in the *zero configuration*. Only one operation can be performed at any given time and a DECR operation is not allowed when the counter is in the zero configuration.

3.4.2 Proof of correct behavior

Does the implementation described above behave correctly? What does it mean here to behave correctly? In other words, what specifications must the implemen-

V_i^t	C_{i-1}^t	B_{i-1}^t	S_{i-1}^t	V_i^{t+1}	C_i^{t+1}	B_i^{t+1}
0	0	0	×	0	0	0
0	0	1	×	1	0	1
0	1	0	×	1	0	0
1	0	1	×	0	0	0
1	0	0	×	1	0	0
1	1	0	0	0	1	0
1	1	0	1	1	0	0

a) $i \geq 0$

S_0^t	C_0^t	S_0^{t+1}
1	0	1
otherwise		0

b)

C_{i+1}^t	C_i^t	C_{i-1}^t	B_i^t	B_{i-1}^t	S_{i+1}^t	S_i^t	S_{i-1}^t	S_i^{t+1}
×	×	1	×	×	×	×	1	1
×	0	×	×	0	×	1	×	1
0	×	×	1	×	1	×	×	1
otherwise								0

c) $i > 0$

Table 3–2: Function $\mathcal{F}$.

tation meet? As mentioned above, the only operations we must be able to perform on the counters are: SET0, INCR, DECR, IF0 and NOP. The specifications may be stated succinctly as follows: provided the capacity of the counter has not been exceeded, an IF0 operation must return the value true if and only if the numbers of DECR operations and INCR operations performed since the last SET0 operation are equal.

Intuitively, one may be easily convinced that our implementation is correct. The concepts of carry save and borrow save vectors are fairly familiar. The one of

significance vector less so. In the remaining of this section, I provide a formal proof of the correctness of my implementation. As is often the case, the proof is rather long for what seems to be, at first sight, obvious enough. I start with a definition followed by a series of eight lemmas.

Definition 3.4.1 *The* value represented in the counter *is given by the the formula:*

$$\sum_{i=0}^{\text{MSB}} (V_i + 2C_i - 2B_i)2^i$$

More specifically, the value represented at some time t is:

$$\sum_{i=0}^{\text{MSB}^t} (V_i^t + 2C_i^t - 2B_i^t)2^i$$

Lemma 3.4.1 *A slice i, $0 < i \leq n$, can hold a carry (borrow) on a given beat only if slice $i - 1$ held one on the previous beat.*

Proof Follows from the definition of $\mathcal{F}$ (table 3–2). ∎

Lemma 3.4.2 *A slice holding a carry (borrow) has its value bit cleared (set).*

Proof Follows from the definition of $\mathcal{F}$. ∎

Lemma 3.4.3 *A slice cannot hold both a carry and a borrow at the same time.*

Proof Follows from lemma 3.4.1 and the fact that we do not allow the INCR and the DECR operations simultaneously. ∎

Lemma 3.4.4 *Two consecutive bit slices cannot both hold carries or both hold borrows.*

Proof By applying lemma 3.4.1, all we need to prove is that a slice cannot generate two consecutive carries or two consecutive borrows. By lemma 3.4.2,

whenever a carry (borrow) is generated in a slice, the V bit of that slice gets the value 0 (1). By the definition of $\mathcal{F}$, a carry (borrow) cannot be generated on the next beat with V at this value. ■

Lemma 3.4.5 *For any i, $0 < i \leq MSB^t$, and any $t > 0$:*

$$V_i^{t+1} + 2C_i^{t+1} - 2B_i^{t+1} = V_i^t + C_{i-1}^t - B_{i-1}^t$$

Proof By lemma 3.4.3 C_{i-1}^t and B_{i-1}^t cannot both have value 1. This leaves 6 possible value combinations for V_i^t, C_{i-1}^t and B_{i-1}^t. By the definition of $\mathcal{F}$, each combination yields a result satisfying the lemma with the exception of $(V_i^t, C_{i-1}^t, B_{i-1}^t) = (1, 1, 0)$ when $S_{i-1}^t = 1$. But in that case, $i > \text{MSB}^t$. ■

The next lemma, although very simple, has an astonishingly long proof.

Lemma 3.4.6 *Any number of beats t after a* SET0 *operation, $B_{\text{MSB}^t}^t = 0$ and except when the counter is in the zero configuration, $C_{\text{MSB}^t}^t + V_{\text{MSB}^t}^t = 1$.*

Proof By induction on t. The conclusion is true for $t = 0$. Let us prove that it also holds for $t + 1$ if it holds for some $t \geq 0$. By the induction hypothesis, at time (on beat) t either the counter is in the zero configuration or $C_{\text{MSB}^t}^t + V_{\text{MSB}^t}^t = 1$ and $B_{\text{MSB}^t}^t = 0$. If the counter is in the zero configuration only an INCR an IF0 or a NOP operation can be performed. In all cases, the hypothesis will still hold on time $t + 1$. If the counter is not in the zero configuration on time t, either $C_{\text{MSB}^t}^t = 1$ or $V_{\text{MSB}^t}^t = 1$. If $C_{\text{MSB}^t}^t = 1$, on the next beat, the MSB slice gets shifted to the left ($\text{MSB}^{t+1} = \text{MSB}^t + 1$) and $V_{\text{MSB}^{t+1}}^{t+1}$ is set to 1 while $C_{\text{MSB}^{t+1}}^{t+1}$ and $B_{\text{MSB}^{t+1}}^{t+1}$ are set to 0 (by the definition of $\mathcal{F}$). Thus, in this case, the hypothesis still holds. The other case ($V_{\text{MSB}^t}^t = 1, C_{\text{MSB}^t}^t = 0$) has two sub-cases depending on whether the MSB slice is in position 0 or not. If the MSB slice is in position 0, it will be affected by the operation performed. A NOP operation will leave it as it is, an INCR operation will

set its C bit and reset its V bit while a DECR operation will put the counter in the zero configuration. In all three cases, the induction hypothesis remains true at time $t+1$. If the MSB slice is not in position 0, it will be affected only by its V bit and the C and B bits of the slice to the right. By lemma 3.4.3, these cannot both be on. If neither one of them are on, the MSB slice remains unchanged and the hypothesis is still true. If $C^t_{\text{MSB}^t-1} = 1$, on time $t+1$, the MSB slice stays in the position it was on time t, C_{MSB} gets set, V_{MSB} gets reset and B_{MSB} is unchanged and the hypothesis is still true. If $B^t_{\text{MSB}^t-1} = 1$, on time $t+1$, the MSB slice gets shifted to the right ($\text{MSB}^{t+1} = \text{MSB}^t - 1$). Here, we have two sub-cases: 1— the new MSB slice is in some position to the left of slice 0; 2— the new MSB slice is in position 0. We consider them in turn. In sub-case 1, the value of the C, V and B bits in the new MSB slice on time $t+1$ (the value of $C^{t+1}_{\text{MSB}^t-1}$, $V^{t+1}_{\text{MSB}^t-1}$ and $B^{t+1}_{\text{MSB}^t-1}$) is a function of $V^t_{\text{MSB}^t-1}$ (the value of the V bit in the new MSB on time t), $C^t_{\text{MSB}^t-2}$ and $B^t_{\text{MSB}^t-2}$ (the values of the C and B bits in the slice to the right of the new MSB slice on time t). Since $B^t_{\text{MSB}^t-1} = 1$, by lemma 3.4.2, $V^t_{\text{MSB}^t-1} = 1$ and by lemma 3.4.4, $B^t_{\text{MSB}^t-2} = 0$. This leaves two possibilities (sub-cases!) depending on the value of $C^t_{\text{MSB}^t-2}$. If this value is 0, the C and B bits in the new MSB slice end up on time $t+1$ with value 0 and the V bit remains with value 1 while if $C^t_{\text{MSB}^t-2} = 1$, the C bit in the new MSB slice takes value 1 and both the V and B bits take value 0. In either cases, the induction hypothesis is preserved. We are now left with a final sub-case to consider, the sub-case when on time t, the MSB slice is in position 1 and slice 0 holds a borrow (sub-case 2). In this sub-case, the second slice to the right of the MSB slice of time t is the virtual slice -1. We can apply exactly the same argument as above (sub-case 1) except that here we may have $B^t_{\text{MSB}^t-2} = 0$. This occurs when we perform a DECR operation on time t. In this case, the counter ends up on time $t+1$ in the zero configuration and the lemma is proved. ■

The following lemma is the most important. It simply states that at any time, the value represented by the counter is the right one.

Lemma 3.4.7 *Provided the capacity of the counter has not been exceeded, the value represented in the counter equals the number of* INCR *operations minus the number of* DECR *operations that have been performed since the last* SET0 *operation.*

Proof I prove by induction on the time (number of beats) t since the last SET0 operation. At time $t = 0$ (just after a SET0 operation), the counter is in the zero configuration and the basis is true. Assuming the hypothesis is true at some time $t > 0$, let us prove that it is also true at time $t+1$. I do so by proving the following equality:

$$\sum_{i=0}^{\mathrm{MSB}^t} (V_i^t + 2C_i^t - 2B_i^t)2^i + \mathrm{OP} = \sum_{i=0}^{\mathrm{MSB}^{t+1}} (V_i^{t+1} + 2C_i^{t+1} - 2B_i^{t+1})2^i \qquad (3.1)$$

where OP $= 1$ if the operation performed on time t is INCR, OP $= -1$ if the operation is DECR and OP $= 0$ otherwise. The summation in the left part of the equation represents the value represented in the counter on time t while the summation in the right part represents the value on time $t+1$.

By the definition of the implementation, we can see the counter as having a virtual slice -1 and consider that $(C_{-1}^t, B_{-1}^t) = (1,0)$ when the operation performed on time t is INCR, that $(C_{-1}^t, B_{-1}^t) = (0,1)$ when it is DECR and that otherwise $(C_{-1}^t, B_{-1}^t) = (0,0)$. We can thus rewrite equation (3.1) as follows:

$$\sum_{i=0}^{\mathrm{MSB}^t} (V_i^t + 2C_i^t - 2B_i^t)2^i + C_{-1}^t - B_{-1}^t = \sum_{i=0}^{\mathrm{MSB}^{t+1}} (V_i^{t+1} + 2C_i^{t+1} - 2B_i^{t+1})2^i \qquad (3.2)$$

Let m denote the minimum of MSB^t and MSB^{t+1} and let us prove the following:

$$\sum_{i=0}^{m}(V_i^t + C_{i-1}^t - B_{i-1}^t)2^i = \sum_{i=0}^{m}(V_i^{t+1} + 2C_i^{t+1} - 2B_i^{t+1})2^i \qquad (3.3)$$

By lemma 3.4.5, each factor of 2^i in the left summation is equal to the corresponding factor in the right summation and thus, the equation is true. Substracting (3.3) from (3.2) we get:

$$\sum_{i=m+1}^{\text{MSB}^t} (V_i^t + 2C_i^t - 2B_i^t)2^i + (2C_m^t - 2B_m^t)2^m = \sum_{i=m+1}^{\text{MSB}^{t+1}} (V_i^{t+1} + 2C_i^{t+1} - 2B_i^{t+1})2^i \quad (3.4)$$

By the definition of $\mathcal{F}$ the MSB position can move by at most one position on each beat. Therefore, only the following three cases are possible:

1. $m = \text{MSB}^t = \text{MSB}^{t+1} - 1$ (the MSB position is shifted to the left)

2. $m = \text{MSB}^t = \text{MSB}^{t+1}$ (the MSB position is unchanged)

3. $m = \text{MSB}^t - 1 = \text{MSB}^{t+1}$ (the MSB position is shifted to the right)

We prove the correctness of equation (3.4) for each case. In case 1, (3.4) becomes:

$$(2C_{\text{MSB}^t}^t - 2B_{\text{MSB}^t}^t)2^{\text{MSB}^t} = (V_{\text{MSB}^t+1}^{t+1} + 2C_{\text{MSB}^t+1}^{t+1} - 2B_{\text{MSB}^t+1}^{t+1})2^{\text{MSB}^t+1} \quad (3.5)$$

By the definition of $\mathcal{F}$, case 1 occurs only when $C_{\text{MSB}^t}^t = 1$. $C_{\text{MSB}^t}^t = 1$ and lemma 3.4.3 implies $B_{\text{MSB}^t}^t = 0$. By the definition of $\mathcal{F}$, $C_{\text{MSB}^t}^t = 1$ implies $V_{\text{MSB}^t+1}^{t+1} = 1$ and $C_{\text{MSB}^t+1}^{t+1} = 0$. Finally, lemma 3.4.6 implies $B_{\text{MSB}^t+1}^{t+1} = 0$ and equation (3.5) is proved. In case 2, (3.4) becomes:

$$(2C_{\text{MSB}^t}^t - 2B_{\text{MSB}^t}^t)2^{\text{MSB}^t} = 0 \quad (3.6)$$

By the definition of $\mathcal{F}$, case 2 occurs only if $C_{\text{MSB}^t}^t = 0$. By lemma 3.4.6, $B_{\text{MSB}^t}^t = 0$ and equation (3.6) is proved. In case 3, equation (3.4) becomes:

$$(V_{\text{MSB}^t}^t + 2C_{\text{MSB}^t}^t - 2B_{\text{MSB}^t}^t)2^{\text{MSB}^t} + (2C_{\text{MSB}^t-1}^t - 2B_{\text{MSB}^t-1}^t)2^{\text{MSB}^t-1} = 0 \quad (3.7)$$

By the definition of $\mathcal{F}$, case 3 occurs only if $C_{\text{MSB}^t} = 0$ and $B_{\text{MSB}^t-1} = 1$. $C_{\text{MSB}^t} = 0$ and lemma 3.4.6 imply that $V_{\text{MSB}^t} = 1$ and $B_{\text{MSB}^t} = 0$. Finally, $B_{\text{MSB}^t-1} = 1$ and lemma 3.4.3 imply that $C_{\text{MSB}^t-1} = 0$ and the correctness of equation (3.7) is proved.

■

Lemma 3.4.8 *When the counter is not in the zero configuration, the value represented in the counter must be greater than 0.*

Proof For a given MSB position, let us find what the smallest value represented in the counter can be. By lemma 3.4.6, either the C bit or the V bit of the MSB slice must be set. Since the C bit has twice the weight of the V bit, the smallest value must be obtained when the V bit is set. Since the B bits have a negative weight, the smallest value must be obtained when the counter has as many and as significant B bits set as possible (in slices 0 to MSB). By lemma 3.4.6, the B bit in the MSB slice cannot be set. By lemma 3.4.4 two consecutive slices cannot both hold borrows. So we will have the most B bits set when the B bits in every other slice, starting from slice MSB $-$ 1 and going to the right, are set. By lemma 3.4.2, when these bits are set so are the V bits in the same slices. Since the V bits have a lesser absolute weight than the B bits, it is still the case that the smallest value will be obtained when those B bits are on. Since the V bits and the C bits have positive weight, we will want to have as few of those on as possible. Assume that the C bits in all the slices are clear and so are the V bits in every other slice starting from slice MSB $-$ 2. In that case, if MSB is even and greater than 0, the value represented in the counter is given by:

$$2^{\text{MSB}} + \sum_{i=1}^{\text{MSB}/2} 2^{2i-1} - 2^{2i} \tag{3.8}$$

The term 2^{MSB} is the contribution of the V bit in the MSB slice, the terms 2^{2i-1} of the summation are the contributions of the other V bits set and the terms 2^{2i} are the contributions of the B bits set. We can apply the following transformations to (3.8):

$$= 2^{\text{MSB}} + \sum_{i=0}^{(\text{MSB}/2)-1} 2^{2i+1} - 2^{2i+2}$$

$$= 2^{\text{MSB}} - 2 \sum_{i=0}^{(\text{MSB}/2)-1} (4^{i})$$

$$= 2^{\text{MSB}} - 2\left(\frac{1 - 4^{\text{MSB}/2}}{1 - 4}\right)$$
$$= \frac{3(2^{\text{MSB}}) + 2 - 2(2^{\text{MSB}})}{3}$$
$$= \frac{2^{\text{MSB}} + 2}{3} \tag{3.9}$$

For any value of MSB even and greater than 0, the formula (3.9) yields a positive value and the lemma is proved for this case. If MSB is odd (and greater than zero), the smallest value that can be represented in the counter is given by:

$$2^{\text{MSB}} + \sum_{i=0}^{(\text{MSB}-1)/2} 2^{2i} - 2^{2i+1} \tag{3.10}$$
$$= 2^{\text{MSB}} - \sum_{i=0}^{(\text{MSB}-1)/2} 2^{2i}$$
$$= 2^{\text{MSB}} - \sum_{i=0}^{(\text{MSB}-1)/2} 4^{i}$$
$$= 2^{\text{MSB}} - \left(\frac{1 - 4^{\frac{\text{MSB}-1}{2}+1}}{1 - 4}\right)$$
$$= \frac{3(2^{\text{MSB}}) + 1 - 2^{\text{MSB}+1}}{3}$$
$$= \frac{2^{\text{MSB}} + 1}{3} \tag{3.11}$$

For any positive odd MSB, the value given by (3.11) is strictly positive and the lemma is proved in that case as well. Only one case is left to consider, the case when MSB = 0. The smallest value represented when MSB = 0 and the counter is not in the zero configuration is when $V_0 = 1$, $C_0 = 0$ and $B_0 = 0$ (by lemma 3.4.6 B_0 must be equal to 0). In that case, the value is 1 and the lemma is proved. ■

Rich with these lemmas, I am now ready to prove that the counter implementation meets its specifications. For this, lemma 3.4.7 almost suffices. It states that after an equal number of INCR and DECR operations, the value represented in the counter is 0. Recall however that the IF0 operation is performed by testing if the counter is in the zero configuration. We thus also need lemma 3.4.8 which explicitly

states that the value 0 can be represented in the counter only when the counter is in the zero configuration.

Theorem 3.4.1 *Provided the capacity of the counter has not been exceeded, if, since the last* SET0 *operation, as many* DECR *operations as* INCR *operations have been performed, the counter must be in the zero configuration*

Proof Follows from lemma 3.4.7 and lemma 3.4.8. ■

Chapter 4

Efficient grammars

4.1 Introduction

As mentioned in the previous chapter, when the grammar used is unambiguous, the number of pointer pairs a processor will need to save during the recognition phase will never exceed the number of non-terminals in the grammar. However large the set of non-terminals is, it will always be finite. This allowed us to conclude that the space complexity of the algorithm is $O(n^2 \log n)$. One of the hidden constants behind this asymptotic measure is specifically the size of the non-terminal set of the grammar. When designing and building actual processors, allocating enough space to hold as many pointer pairs as there are non-terminals could prove prohibitive. In this chapter, I take a look at grammar properties that could ensure that the number of pointers the processors could need to save is low. I say that a given grammar is *k-efficient* if for this grammar, this number is bounded by k. A 1-efficient grammar, I simply call an *efficient* grammar. I first show that G_1 (see page 22), the grammar used in examples in the last two chapters, is efficient. I then exhibit two distinct sufficient conditions for a grammar to be efficient. I provide counter examples to

show that these are not necessary conditions. I also provide a counter example to show that unambiguity is not a necessary condition either. I conclude with a small discussion on *structural ambiguity* and decidability.

4.2 G_1 is efficient

To prove G_1 efficient, I prove a series of lemmas. The first three of these are auxiliary lemmas stating two properties of strings derivable from the non-terminals E, T and F or from the right-hand sides $E\underline{)}$, $\underline{+}T$ and $\underline{*}F$. Each of the others states that any string reducible to a given right-hand side of G_1 with two non-terminals is reducible only to this right-hand side and only according to one partitioning. The efficiency of G_1 follows immediately from these.

To express the fact that a string contains as many left and right parentheses I will say that the string is *()-balanced* (parenthesis balanced). If it contains more left (or right) parentheses then right (or left) parentheses, I will say that it is *(-heavy* (left parenthesis heavy) (or *)-heavy* (right parenthesis heavy)). Notice that a string must be either (-heavy, ()- balanced or)-heavy.

Lemma 4.2.1 *Any string reducible to either E, T or F is of odd length.*

Proof By induction on the length of the string. The basis is true for the only string of length 1 reducible to E, T or F, namely a. Suppose that the hypothesis

is true for any string shorter than some string s of length greater than one. If s reduces to either E, T or F, then, one of the following must be true:

1. $E_{+}T \Rightarrow E\underline{+}T \stackrel{*}{\Rightarrow} s_1 + s_2 = s$,
2. $T_{*}F \Rightarrow T\underline{*}F \stackrel{*}{\Rightarrow} s_1 * s_2 = s$,
3. $\underline{(}E_{)} \Rightarrow \underline{(}E\underline{)} \stackrel{*}{\Rightarrow} (s_1) = s$.

The induction hypothesis holds for s_1 (and s_2), it thus also holds for s. ∎

Lemma 4.2.2 *Any string reducible to either* $E\underline{)}$, $\underline{+}T$ *or* $\underline{*}F$ *is of even length.*

Proof Follows from lemma 4.2.1 and the fact that each of the non-terminals $\underline{)}$, $\underline{+}$ and $\underline{*}$ directly derive a terminal. ∎

Lemma 4.2.3 *Any string s reducible to either E, T or F is ()-balanced and any prefix of that string is either (-heavy or ()-balanced.*

Proof By induction on the length of the string. The basis is true for the only string of length 1 reducible to E, T or F, namely a. Suppose the hypothesis is true for any string shorter than some string s of length greater than 1. If s reduces to either E, T or F then one of the following is true:

1. $E_{+}T \Rightarrow E\underline{+}T \stackrel{*}{\Rightarrow} s_1 + s_2 = s$,
2. $T_{*}F \Rightarrow T\underline{*}F \stackrel{*}{\Rightarrow} s_1 * s_2 = s$,
3. $\underline{(}E_{)} \Rightarrow \underline{(}E\underline{)} \stackrel{*}{\Rightarrow} (s_1) = s$.

The induction hypothesis holds for s_1 (and s_2), it thus also holds for s. ∎

Lemma 4.2.4 *Any string reducible to the right-hand side $E\,_+T$ is reducible only to this right-hand side and only according to one partitioning (i.e. $E\,_+T \stackrel{*}{\Rightarrow} s_1s_2 = s$, $E \stackrel{*}{\Rightarrow} s_1$, $_+T \stackrel{*}{\Rightarrow} s_2$, $A\,B \stackrel{*}{\Rightarrow} s_3s_4 = s$, $A \stackrel{*}{\Rightarrow} s_3$, $B \stackrel{*}{\Rightarrow} s_4$ implies $A = E$, $B =_+ T$, $s_1 = s_3$ and $s_2 = s_4$).*

Proof Since $E\,_+T \stackrel{*}{\Rightarrow} s$, s is of an odd length at least 3. Thus none of the single terminal right-hand sides of G_1 and none of the right-hand sides mentioned in lemma 4.2.2 can derive s. Since $E\,_+T \stackrel{*}{\Rightarrow} s$, s must contain a $+$. Suppose $(\,E_)\, \stackrel{*}{\Rightarrow} s$. Then s must be of the form $(u+v)$ where $E \stackrel{*}{\Rightarrow} u+v$, $(u = s_1$ and $+v) = s_2$. By lemma 4.2.3 any prefix of $u+v$ is either (-heavy or ()-balanced since $E \stackrel{*}{\Rightarrow} u+v$. Since $E \stackrel{*}{\Rightarrow} s_1 = (u$, by the same lemma $(u$ is ()-balanced. It follows that u, a prefix of $u+v$ is)-heavy. A string cannot be)-heavy and at the same time be either (-heavy or ()-balanced. We have a contradiction and conclude that $(\,E_)\, \stackrel{*}{\not\Rightarrow} s$.

Suppose $T\,_*F \stackrel{*}{\Rightarrow} s$. Then one of the following must be true:

1. $s = t + u * v$, $t = s_1$, $+u * v = s_2$, $T \stackrel{*}{\Rightarrow} t + u$ and $_*F \stackrel{*}{\Rightarrow} *v$;

2. $s = t * u + v$, $t * u = s_1$, $+v = s_2$, $T \stackrel{*}{\Rightarrow} t$ and $_*F \stackrel{*}{\Rightarrow} *u + v$.

Let us look at the first case. The terminal $+$ is accessible to the non-terminal T only via the right-hand side $(\,E_)$. Thus $T \stackrel{*}{\Rightarrow} t+u$ implies that $t+u$ must be of the form $t_1(t_2+u_1)u_2$ where $E \stackrel{*}{\Rightarrow} t_2+u_1$, $t_1(t_2 = t$ and $u_1)u_2 = u$. Since $E \stackrel{*}{\Rightarrow} s_1 = t_1(t_2$, by lemma 4.2.3, $t_1(t_2$ is ()-balanced and any prefix of $t_1(t_2$ must either be (-heavy or ()-balanced. Thus t_1 must be (-heavy and t_2 must be)-heavy. But t_2 is a prefix of t_2+u_1 and by the same lemma t_2 cannot be)-heavy since $E \stackrel{*}{\Rightarrow} t_2+u_1$. We have a contradiction and conclude that case 1 cannot be. Let us look at case 2. The terminal $+$ is accessible to the non-terminal $_*F$ only via the right-hand side $(\,E_)$. Thus $_*F \stackrel{*}{\Rightarrow} *u+v$ implies that $*u+v$ is of the form $*u_1(u_2+v_1)v_2$ where

$E \stackrel{*}{\Rightarrow} u_2 + v_1$, $u_1(u_2 = u$ and $v_1)v_2 = v$. Since $E \stackrel{*}{\Rightarrow} s_1 = t * u$, by lemma 4.2.3, $t * u_1(u_2$ is ()-balanced and $t * u_1($ is (-heavy. Thus u_2 is)-heavy. But u_2 is a prefix of $u_2 + v_1$ and by the same lemma u_2 is either ()-balanced or (-heavy since $E \stackrel{*}{\Rightarrow} u_2 + v_1$. We have a contradiction and conclude that case 2 cannot be. We conclude that $T\,_*F \stackrel{*}{\not\Rightarrow} s$.

Suppose now that $pq = s, E \stackrel{*}{\Rightarrow} p$, $_+T \stackrel{*}{\Rightarrow} q$, $p \neq s_1$ and $q \neq s_2$. Then one of the following must be true:

1. $s = t + u + v$, $s_1 = t$, $s_2 = +u + v$, $p = t + u$ and $q = +v$;

2. $s = t + u + v$, $s_1 = t + u$, $s_2 = +v$, $p = t$ and $q = +u + v$.

Let us look at the first case. Since + is accessible to T only via the right-hand side (E), $_+T \stackrel{*}{\Rightarrow} s_2 = +u + v$ implies that $+u + v$ is of the form $+u_1(u_2 + v_1)v_2$ where $E \stackrel{*}{\Rightarrow} u_2 + v_1$, $u_1(u_2 = u$ and $v_1)v_2 = v$. Since $E \stackrel{*}{\Rightarrow} p = t + u = t + u_1(u_2$, by lemma 4.2.3 $t + u_1(u_2$ is ()-balanced. Therefore $t + u_1($ is (-heavy and u_2 is)-heavy. But u_2 is a prefix of $u_2 + v_1$ and by the same lemma it is either ()-balanced or (-heavy. We have a contradiction and conclude that case 1 above cannot be. Case 2 is dual to case 1 and by the same argument we conclude that it cannot be. We have covered every right-hand side of the grammar and the proof is complete. ∎

Lemma 4.2.5 *Any string reducible to the right-hand side $T\,_*F$ is reducible only to this right-hand side and only according to one partitioning (i.e. $T\,_*F \stackrel{*}{\Rightarrow} s_1 s_2 = s$, $T \stackrel{*}{\Rightarrow} s_1$, $_*F \stackrel{*}{\Rightarrow} s_2$, $AB \stackrel{*}{\Rightarrow} s_3 s_4 = s$, $A \stackrel{*}{\Rightarrow} s_3$, $B \stackrel{*}{\Rightarrow} s_4$ implies $A = T$, $B =_* F$, $s_1 = s_3$ and $s_2 = s_4$).*

Proof Since $T\,_*F \stackrel{*}{\Rightarrow} s$, s is of an odd length at least 3. Thus none of the single terminal right-hand sides of G_1 and none of the right-hand sides mentioned in lemma 4.2.2 can derive s. By lemma 4.2.4, $E\,_+T \stackrel{*}{\not\Rightarrow} s$ since $T\,_*F \stackrel{*}{\Rightarrow} s$. Because

$T{}_*F \overset{*}{\Rightarrow} s$, s must contain a $*$. Suppose $\underline{(}E) \overset{*}{\Rightarrow} s$ then $s = s_1s_2$ must be of the form $(u * v)$ where $E \overset{*}{\Rightarrow} u * v$, $(u = s_1$ and $*v) = s_2$. By lemma 4.2.3, u, a prefix of $u * v$ can only be either ()-balanced or (-heavy since $E \overset{*}{\Rightarrow} u * v$. Since $T \overset{*}{\Rightarrow} s_1 = (u$, by lemma 4.2.3 $(u$ is ()-balanced and thus, u is)-heavy. We have a contradiction and conclude that $\underline{(}E) \not\overset{*}{\Rightarrow} s$. We have covered every right-hand side of G_1 besides $T{}_*F$. All we need to consider now is the right hand side $T{}_*F$ and a different partitioning of s. Suppose that $pq = s$, $T \overset{*}{\Rightarrow} p$, ${}_*F \overset{*}{\Rightarrow} q$, $p \neq s_1$ and $q \neq s_2$. Then one of the following must be true:

1. $s = t * u * v$, $s_1 = t$, $s_2 = *u * v$, $p = t * u$ and $q = *v$;

2. $s = t * u * v$, $s_1 = t * u$, $s_2 = *u$, $p = t$ and $q = *u * v$.

One case being the dual of the other, we only need to consider one. Let us look at 1. Since $*$ is accessible to F only via the right hand side $\underline{(}E)$, ${}_*F \overset{*}{\Rightarrow} s_2 = *u * v$ implies that $*u * v$ is of the form $*u_1(u_2 * v_1)v_2$ where $E \overset{*}{\Rightarrow} u_2 * v_1$, $u_1(u_2 = u$ and $v_1)v_2 = v$. Since $T \overset{*}{\Rightarrow} p = t * u = t * u_1(u_2$, by lemma 4.2.3 $t * u_1(u_2$ is ()-balanced and thus $t * u_2($ is (-heavy and u_2 is)-heavy. But u_2 is a prefix of $u_2 * v_1$ and by the same lemma u_2 is either ()-balanced or (-heavy. We have a contradiction and conclude that case 1 (and consequently case 2) cannot be. ∎

Lemma 4.2.6 *Any string reducible to the right-hand side $\underline{(}E)$ is reducible only to this right-hand side and only according to one partitioning (i.e. $\underline{(}E) \overset{*}{\Rightarrow} s_1s_2 = s$, $\underline{(} \overset{*}{\Rightarrow} s_1$, $E) \overset{*}{\Rightarrow} s_2$, $AB \overset{*}{\Rightarrow} s_3s_4 = s$, $A \overset{*}{\Rightarrow} s_3$, $B \overset{*}{\Rightarrow} s_4$ implies $A = \underline{(}$, $B = E)$, $s_1 = s_3$ and $s_2 = s_4$.*

Proof Since $E \overset{*}{\Rightarrow} s$, s must be an odd length at least 3. Thus, none of the single terminal right-hand sides of G_1 and none of the right-hand sides mentioned in lemma 4.2.2 can derive s. By lemma 4.2.4 and 4.2.5 none of the right-hand sides $E{}_+T$ or $T{}_*F$ can derive s since $\underline{(}E \overset{*}{\Rightarrow} s$. We have covered every right-hand side

of G_1 besides $\underline{(}E)$. All we need to consider is the possibility of a reduction according to a different partitioning of s. But the only rule of G_1 with $\underline{(}$ as a left hand side is $\underline{(} \rightarrow ($ so the left partition of s can only be the one-token string (. ∎

Lemma 4.2.7 *Any string reducible to either of the right-hand sides $E\underline{)}$, $\underline{+}T$ or $\underline{*}F$ is reducible only to this right-hand side and only according to one partitioning.*

Proof By lemma 4.2.2, any string s reducible to either of these right-hand sides must be of even length and thus s cannot be reducible to any of the right-hand sides $E_{+}T$, $T_{*}F$ and $\underline{(}E)$. The right-hand sides $\underline{+}T$ and $\underline{*}F$ cannot both derive the same string since the former can only derive strings starting with the symbol $+$ while the latter can only derive strings starting with the symbol $*$. Suppose a string s was reducible to both right-hand sides $E\underline{)}$ and $\underline{+}T$. Then s would be of the form $+t)$ where $E \stackrel{*}{\Rightarrow} +t$ and $T \stackrel{*}{\Rightarrow} t)$. By lemma 4.2.3, t would have to be at the same time (-heavy and ()-balanced. This is impossible and we conclude that s cannot be reducible to both $E\underline{)}$ and $\underline{+}T$. By a similar argument it follows that no string can be reducible to both $E\underline{)}$ and $\underline{*}F$. Trivially, a string reducible to either of the right-hand side $E\underline{)}$, $\underline{+}T$ or $\underline{*}F$ is reducible to this right-hand side only according to one partitioning. ∎

Theorem 4.2.1 *G_1 is 1-efficient.*

Proof By lemmas 4.2.4, 4.2.5, 4.2.6 and 4.2.7 any string s reducible to any two non-terminals right-hand side of G_1 is reducible according to only one partitioning thus G_1 is efficient. ∎

4.3 NT-disjunction and RHS-disjunction

The whole of the previous section has been devoted to show that a very simple grammar, G_1, is efficient. Is it possible to find a property that characterizes efficient grammars? My search for such a characterization has led me to consider two properties which I call *non-terminal disjunction* and *right-hand side disjunction.* I show that each of these properties in conjunction with unambiguity is a sufficient condition for efficiency but not a necessary condition.

A grammar is non-terminal disjoint if distinct non-terminals of the grammar derive only distinct strings of terminals.

Definition 4.3.1 *A grammar* $G = \langle \Sigma, N, P, S \rangle$ *is* non-terminal disjoint (nt-disjoint) *if for any* $A, B \in N$, $s \in \Sigma^*$, $A \stackrel{+}{\Rightarrow} s$, $B \stackrel{+}{\Rightarrow} s$ *implies* $A = B$.

Note that since it is undecidable whether the intersection of two CFGs is empty or not [Aho 72], it is undecidable whether two given non-terminals of a grammar can derive the same string or not. This does not necessarily imply that grammar *nt*-disjunction is undecidable. At the end of this section, I prove the undecidability of *nt*-disjunction in the case of general (non CNF) CFGs. In the case of grammars in Chomsky normal form, the question remains open.

Theorem 4.3.1 *Any grammar G that is unambiguous and* nt-*disjoint is also efficient.*

Proof Let us consider a string s reducible to some non-terminal A of G. Because G is *nt*-disjoint, s reduces to no other terminal. Now, suppose s is a substring of r, a sentence of G, that the derivation from A to s is part of the derivation from the distinguished symbol to r, and that s is reducible to A according to more than one

partitioning. Then there exist at least two parses for r, one for each partitioning of s. But G is unambiguous so there can only exist one parse for r. We have a contradiction and conclude that s reduces to A according to only one partitioning which implies that G is efficient. ∎

G_1 provides us with a counter example to prove that *nt*-disjunction is not a necessary condition for grammar efficiency.

Theorem 4.3.2 *An unambiguous efficient grammar G may not be* nt-*disjoint.*

Proof The sentence a reduces to F, to T and to E so G_1 is not *nt*-disjoint and according to theorem 4.2.1, G_1 is efficient. ∎

What we can observe is that in the case of G_1 when a string reduces to more than one non-terminal, it does so according to only one right-hand side. This raises the question as to whether it would not be more appropriate to consider the *disjunction* at the level of right-hand sides rather then that of non-terminals.

Definition 4.3.2 *A grammar $G = \langle \Sigma, N, S, P \rangle$ is* right-hand side disjoint *if for any $A \rightarrow A_1 A_2$, $B \rightarrow B_1 B_2 \in P$, $s \in \Sigma^*$, $A_1A_2 \stackrel{+}{\Rightarrow} s$, $B_1B_2 \stackrel{+}{\Rightarrow} s$ implies $A_1A_2 = B_1B_2$.*

Like *nt*-disjunction, *rhs*-disjunction is a sufficient but not a necessary condition for efficiency. Also, like *nt*-disjunction, *rhs*-disjunction is undecidable in the case of general CFGs.

Theorem 4.3.3 *Any grammar G that is unambiguous and* rhs-*disjoint is efficient.*

Proof Let us consider a string s reducible to some non-terminal A via some rule $A \rightarrow B\,C$ of G. Because G is *rhs*-disjoint, s may reduce to another non-terminal

but only via the right-hand side $B\,C$. Now, suppose s is a substring of r, some sentence of G, that the derivation from the right-hand side $B\,C$ to s is part of the derivation from the distinguished symbol to r and that s is reducible (to $B\,C$) according to more than one partitioning of s. Then there must exist at least two parses for r, one for each partitioning of s. But G is unambiguous so there can only exist one parse for r. We have a contradiction and conclude that s is reducible to $B\,C$ according to only one partitioning. Thus G is efficient. ∎

In preparation for the proof of the non-implication of *rhs*-disjunction from efficiency, I introduce the following grammar:

$$G_2 = \langle \{a, b, c\}, \{A, B, D, F, I, J, S\}, P, S \rangle$$

where P is:

$S \rightarrow A\,D$	$D \rightarrow F\,D$	$I \rightarrow J\,I$
$S \rightarrow B\,I$	$D \rightarrow c$	$I \rightarrow c$
$A \rightarrow a$	$F \rightarrow c$	$J \rightarrow c$
$B \rightarrow b$		

G_2 generates the regular set $(ac^+ \mid bc^+)$. Its right-hand sides $F\,D$ and $J\,I$ both derive the strings in the regular set (c^+) so the grammar is not *rhs*-disjoint. However, it is unambiguous and efficient. Of these four assertions, the first two can easily be seen to be true while one may wish to be convinced of the others.

Lemma 4.3.1 *G_2 is unambiguous.*

Proof Suppose s is a string of $L(G_2)$. Let us prove that the rules composing a leftmost parse of s are all determined uniquely. Since A derives only the string 'a', B derives only the string 'b' and the only alternatives for S are $A\,D$ and $B\,I$, s must start with either an 'a' or a 'b'. If it starts with an a, the first two rules of the leftmost parse must be $S \rightarrow A\,D$ and $A \rightarrow a$ while if it starts with a 'b'

they must be $S \rightarrow B\,I$ and $B \rightarrow b$. In the former case, the rest of the string must be derived from D. Since the only terminal accessible from D is 'c' and the grammar has no empty production, the rest of the string must consist of one or more symbols 'c'. If the rest of the string contains only one 'c' the next and last rule can only be $D \rightarrow c$ since the only other alternative of D, namely $F\,D$, has at least two non-terminals. If it contains more than one 'c' then the next two rules can only be $D \rightarrow F\,D$ and $F \rightarrow c$. In this last case, the rest of the string s after the first 'c' must be derived from D. We can apply the above argument recursively and conclude that rules composing the left sub-parse of this string of 'c's will be determined uniquely. In the case that s starts with a 'b', exactly the same situation prevails but with non-terminal D replaced by non-terminal I and non-terminal F replaced by non-terminal J. Since in all cases, the rules of a leftmost parse of s are determined uniquely, we conclude that G_2 is unambiguous. ■

Lemma 4.3.2 *G_2 is efficient.*

Proof A string reducible to S must consist of either an 'a' or a 'b' followed by one or more 'c's. In either case, the left partition of the string must consist of its first symbol and the right partition must consist of the rest of the string. This is because $A\,D$ and $B\,I$ are the only alternatives of S and A and B derive only the one symbol strings 'a' and 'b' respectively. Also, no other non-terminal of G_2 can derive a string reducible to S. The only string reducible to either F or J is the one symbol string 'c'. Both non-terminals D and I can derive strings of one or more 'c's and only those strings. A string of more than one 'c' is reducible to D and I only according to the partitioning consisting of a left partition composed of the first 'c' of the string and a right partition composed of the remainder. We have shown that for any string reducible to some non-terminal of G_2, the final reduction, to whatever non-terminal, involves a unique partitioning of the string. We thus conclude that G_2 is efficient. ■

We now have the necessary elements to prove the following theorem on *rhs*-disjunction.

Theorem 4.3.4 *A grammar that is unambiguous and efficient may not be* rhs-*disjoint.*

Proof G_2 provides the necessary counter-example. By lemmas 4.3.1 and 4.3.2, grammar G_2 is unambiguous and efficient. Since the string 'cc' is reducible to $D\,F$ as well as to $I\,J$, two right-hand sides of G_2, G_2 is not *rhs*-disjoint. ∎

4.3.1 Undecidability of nt/rhs-disjunction

I prove that for general (non CNF) grammars, *nt*-disjunction and *rhs*-disjunction are two undecidable problems. This means that no algorithm exists that can, when given any arbitrary grammar G, output whether G is *nt*-disjoint (*rhs*-disjoint) or not. The proofs of this resemble very much the proof of the undecidability of grammar ambiguity found in [Aho 72]. It resorts to the classic strategy of reducing Post's Correspondence Problem (PCP) to the problem we want to show undecidable. Let us call the latter P. To reduce PCP to P, we must find a transformation that maps instances of PCP to instances of P. The transformation must be such that the transformed instance (of P) is to have (or not have) a solution if and only if the original instance (of PCP) has one. The argument then brought forward is that if an algorithm existed for P the composition of the transformation with this algorithm would provide us with an algorithm for PCP. But since PCP is known to be undecidable such an algorithm cannot exist and therefore, one cannot exist for P either. (Note that one does not have to resort to PCP specifically. Any undecidable problem will do.)

For the purpose of introducing my notation, I reproduce the definition of PCP.

Definition 4.3.3 *An instance of* Post's Correspondence Problem *is a three-tuple* $\langle \Sigma, W, X \rangle$ *where* W *and* X *are two equal length lists of strings over the alphabet* Σ.

$$W = w_1, w_2, \dots w_n$$
$$X = x_1, x_2, \dots x_n$$

The instance has a solution if for some sequence of integers $i_1, i_2, \dots i_m$, $m \geq 1$,

$$w_{i_1} w_{i_2} \dots w_{i_m} = x_{i_1} x_{i_2} \dots x_{i_m}.$$

In such a case, the sequence of integers $i_1, i_2, \dots i_m$, $m \geq 1$ *constitutes a solution of the instance.*

For showing grammar *nt*-disjunction undecidable, I shall use a transformation which from any instance C of PCP creates a grammar G_C. The grammar G_C is *nt*-disjoint if and only if C has no solution. This transformation is only slightly different from the transformation exhibited in [Aho 72] for showing grammar ambiguity undecidable.

Theorem 4.3.5 *General CFG* nt*-disjunction is undecidable*

Proof From some arbitrary instance of PCP $C = \langle \Sigma, (w_1, \dots w_n), (x_1, \dots x_n) \rangle$ let us create the grammar $G_C = \langle \Sigma \cup R, \{S, S_W, S_X\}, P, S \rangle$ where R is a set of symbols $\{\flat, r_1, \dots r_n\}$ disjoint from Σ and where P contains the rules $S \rightarrow \flat S_W$, $S \rightarrow \flat S_X$. For each i, $1 \leq i \leq n$, P also contains the rules $S_W \rightarrow w_i S_W r_i$, $S_W \rightarrow w_i r_i$, $S_X \rightarrow x_i S_X r_i$ and $S_X \rightarrow x_i r_i$. Let us denote by L_W and L_X respectively the sets of strings of terminals derivable from S_W and S_X. It is easy to see that $L_W = \{w_{i_1} \dots w_{i_m} r_{i_m} \dots r_{i_1} \mid m \geq 1\}$ and $L_X = \{x_{i_1} \dots x_{i_m} r_{i_m} \dots r_{i_1} \mid m \geq 1\}$. The instance C has a solution if and only if $L_W \cap L_X \neq \emptyset$. Indeed, suppose $i_1 \dots i_m, m \geq 1$ is a solution for C, then the string $w_{i_1} \dots w_{i_m} r_{i_m} \dots r_{i_1}$ in L_W equals the string $x_{i_1} \dots x_{i_m} r_{i_m} \dots r_{i_1}$ in L_X and thus, $L_W \cap L_X \neq \emptyset$. Conversely,

suppose a string $w_{i_1} \dots w_{i_m} r_{i_m} \dots r_{i_1}$ in L_W is also in L_X then this string must be $x_{i_1} \dots x_{i_m} r_{i_m} \dots r_{i_1}$ and the sequence $i_1 \dots i_m$ must be a solution for C. Grammar G_C has only the three non-terminals S, S_W and S_X. All strings of terminals derivable from S must begin with the symbol $\flat$ while no strings derivable from S_W or S_X begin with this symbol. Therefore, if G_C is to have two non-terminals that derive the same string, those could only be S_W and S_X. It follows that G_C can be *nt*-disjoint if and only if C has no solution. ∎

Using the same transformation as in the theorem above, I can also prove the undecidability of *rhs*-disjunction.

Theorem 4.3.6 *General CFG* rhs-*disjunction is undecidable*

Proof From an arbitrary instance of PCP $C = \langle \Sigma, (w_1, \dots w_n), (x_1, \dots x_n) \rangle$ let us create the grammar $G_C = \langle \Sigma \cup R, \{S, S_W, S_X\}, P, S \rangle$ in the same way as in theorem 4.3.5. The only other right-hand side that could derive a string derivable from the right-hand side $\flat\, S_W$ is $\flat\, S_X$ (and vice versa) since the terminal $\flat$ is accessible from no other right-hand side. $\flat\, S_W$ and $\flat\, S_X$ can derive the same string if and only if S_W and S_X can derive the same string and thus if and only if C has a solution. All the other right-hand sides of G_C end with a terminal symbol from R. Let us consider some right-hand side $w_i r_i$, $1 \leq i \leq n$. Only right-hand sides ending with r_i could derive the string $w_i r_i$. The right-hand sides $w_i\, S_W\, r_i$ and $x_i\, S_X\, r_i$ cannot derive this string since they introduce at least one other symbol from R. The other right-hand side ending with r_i, $x_i r_i$, will derive the string $w_i r_i$ if and only if C has a solution, namely i. Let us now consider the right-hand side $w_i\, S_W\, r_i$. From the previous argument, it follows that only the right-hand side $x_i\, S_X\, r_i$ could derive a string derivable from $w_i\, S_W\, r_i$. That could happen if and only if S_W and S_X can derive a common string and thus if and only if C has a solution. The argument above generalizes to the other right-hand sides

of G_C. We conclude that G_C is *rhs*-disjoint if and only if C has no solution and thus that *rhs*-disjunction is undecidable. ■

Note that the fact that *nt*-disjunction and *rhs*-disjunction are undecidable for general CFGs does not imply that the same is true in the case of grammars in CNF. The CNF restriction may very well make both problems decidable but this is not yet known.

4.4 Efficiency, ambiguity and structural ambiguity

In this chapter I have only considered unambiguous grammars. The unambiguity property played an important part in the proofs of theorems 4.3.1 and 4.3.3. One may wonder whether or not the unambiguity property would not itself be a necessary condition for grammar efficiency? The answer to that question is no.

Theorem 4.4.1 *A grammar can be ambiguous and efficient.*

Proof Consider the following simple grammar:

$$G_3 = \langle \{a\}, \{A, B, S\}, \{\, S \rightarrow A\,B\,,\; S \rightarrow B\,A\,,\; A \rightarrow a\;,\; B \rightarrow a\;\}, S\rangle$$

G_3 generates only the string aa. It can generate it however in two different ways so it is ambiguous. But aa can be partitioned into two non-empty sub-strings in only one way so G_3 is necessarily efficient. ■

"But $L(G_3)$ is finite," one might say. Are there ambiguous efficient grammars whose language are infinite? Yes. Consider:

$$G_4 = \langle \{a\}, \{A, B, C, S\}, P, S \rangle$$

where P is:

$$\begin{array}{ccc} S \rightarrow A\,B & A \rightarrow A\,B & B \rightarrow a \\ S \rightarrow A\,C & A \rightarrow a & C \rightarrow a \\ S \rightarrow a & & \end{array}$$

$L(G_4)$ is the regular set a^+. Any sentence a^+a has two parses, one that includes the rule $S \rightarrow A\,B$ and the other that includes the rule $S \rightarrow A\,C$, so G_4 is ambiguous. On the other hand, only one partitioning of some string $a^n a$, $n \geq 1$, namely $a^n \mid a$, can be involved in a reduction of this string to a non-terminal (A or S) and so G_4 is efficient.

The fact that efficiency does not imply unambiguity is not a totally new result. Graham and Harrison [Graham 76a] have pointed out a very similar fact in relation with what they call *structurally unambiguous* grammars. A structurally unambiguous sentence is one that may have more than one parse tree but whose parse trees all have the same *shape*. A grammar is structurally unambiguous if all its sentences are structurally unambiguous. (A grammar (sentence) is structurally ambiguous if it is not structurally unambiguous.) Graham and Harrison exhibited a grammar very similar to G_3 to show that a structurally unambiguous grammar can be ambiguous (G_3 itself actually shows this). Structural unambiguity and efficiency may, at first, look to be the same thing. While it is certainly the case that any efficient grammar is structurally unambiguous, the converse is not true. Consider:

$$G_5 = \langle\{a, b\}, \{A, B, C, D, E, F\}, P, S\rangle$$

where P is:

$$\begin{array}{lll} S \rightarrow A\,C & S \rightarrow E\,B & A \rightarrow a \\ C \rightarrow D\,B & E \rightarrow A\,F & B \rightarrow b \\ D \rightarrow A\,A & F \rightarrow A\,B & \end{array}$$

G_5 generates the two strings aaab and aabb unambiguously and is thus structurally unambiguous. But it is not efficient because the string aab can be reduced to C according to one partitioning, aa | b, and to E according to another, a | ab.

4.5 A lot of open questions

In this chapter, I have been looking for a characterization of efficient grammars. I have exhibited two conditions that are sufficient but not necessary for efficiency. My search for such a characterization has been unsuccessful. In the circumstances, a question that naturally arises is the one of decidability. Is grammar efficiency decidable? I would guess it is. Unfortunately again, I have not yet been able to prove it. My results do not converge to some neat solid conclusion. This may be somewhat unsatisfactory. On the positive side, I think the search has pointed out a few interesting open questions: "how can we characterize efficient grammars?", "is grammar efficiency decidable?", "is *nt*-disjunction of grammars in CNF decidable?" and "is *rhs*-disjunction of grammars in CNF decidable?".

Chapter 5

Outputting multiple parses

5.1 Introduction

When describing my extension in chapter 3, I assumed that the problem to be solved either had only one solution or that, if it had many, any one of them would do and so one could be chosen arbitrarily. While for problems like the building of optimal binary search trees or the finding of the optimal order for multiplying matrices, this approach will nearly always be satisfactory such may not be the case for other problems, like CFL parsing. If we are analysing strings with an ambiguous grammar, we will often want to obtain all the parses of an input string (and possibly apply thereafter some criteria to select one parse among them). This chapter is devoted to the presentation of two different methods by which we can have our parsing algorithm (extension to K-GKT) output multiple parses. The first method adds a factor n to the space complexity of the extension while the second keeps the original space complexity. Before presenting the algorithm, let me discuss a few related issues.

5.2 Preliminaries

5.2.1 Partitioning/right-hand side pairs

If a string s of length greater then one reduces to some non-terminal L ($L \stackrel{*}{\Rightarrow} s$) then for some rule $L \rightarrow R_1\, R_2$ of the grammar and a partitioning $s_1|s_2$ of s we have $R_1 \stackrel{*}{\Rightarrow} s_1$ and $R_2 \stackrel{*}{\Rightarrow} s_2$. In the following I shall often talk about the right-hand side of the rule and the string partitioning involved in this way in a reduction as a pair. I shall refer to such a pair by the term *partitioning/right-hand side* pair which I abbreviate to *p/rhs* pair. I shall also often use the abbreviation *rhs* instead of the term *right-hand side* (of a rule). During the recognition phase, when a processor finds the first non-terminal of the rhs of some rule in the set associated with a left partition of the string it spans and finds the second non-terminal of the rhs in the set of the corresponding right partition, it inserts in its set the left-hand side non-terminal of the rule. It also records, for use in the marking and the output phases, the rule and the pair of counter values that point to the processors involved. Actually, the left-hand side of the rule need not be recorded explicitly since it can be recovered from the rhs and the grammar. Observe also that the pointer pair saved corresponds to the relevant partitioning of the string spanned. Hence, the information saved is directly related to a p/rhs pair. In the following I shall actually refer to this information as a p/rhs pair. I shall refer to the fact that a processor is allowed to record a p/rhs by saying that the processor *deduces* or simply *finds* the p/rhs pair.

The number of p/rhs pairs

If the grammar is unambiguous then for each string there can be no more than one p/rhs pair involved in the reduction of the string to a given non-terminal. For an

ambiguous grammar this is obviously not the case. A question of interest to us is how many p/rhs pairs could lead to the reduction of a string to some non-terminal? (It is of interest to us because our processors will need to record these pairs.) It is easy to show that in some cases all the partitionings of a string could be involved in a reduction of the string to a non-terminal. Consider the following grammar:

$$G_6 = \langle \{a\}, \{A\}, \{\, A \rightarrow A\,A\,,\; A \rightarrow a\,\}, A \rangle$$

All the $n - 1$ partitionings of a string a_n can lead to the reduction of the string, via the rule $A \rightarrow A\,A$, to the non-terminal A. There could certainly not be more p/rhs pairs associated with a string than the number of partitionings of the string times the number of rhs in the grammar. This second factor is a constant and so the number of p/rhs that could be involved in reductions of the string is bounded by a value of $O(n)$.

Observe that a rhs can appear in more than one rule. If a p/rhs pair leading to the reduction of a string to a non-terminal contains such a right-hand side then this pair also leads to the reduction of the string to the other left-hand side non-terminals that share this rhs. There is no need however to have the processor store the pair more than once.

5.2.2 Parses and sub-parses

A parse, recall, is a sequence of rules that can be applied to derive a sentence from the distinguished symbol. A parse is composed of sub-parses. A sub-parse is a sequence of rules that can be applied to derive a segment of a sentence from a given non-terminal. I will often refer to this non-terminal as the non-terminal to which the sub-parse is *relative*. Consider a sub-parse of a given parse. If there is a second sub-parse relative to the same non-terminal that produces the same segment then we can replace the first sub-parse by the second to obtain a new parse. A sub-parse

may itself be composed of sub-parses so the comment I have just made applies recursively to sub-parses. The two algorithms that I present in this chapter both implement some sort of recursive enumeration of the parses of the input.

5.3 Recursive enumeration: the $O(n^3 \log n)$ method

I now describe one method for outputting the successive parses of an input string. I assume that each processor has enough storage to hold all the p/rhs it may ever deduce and that a processor can access the p/rhs pairs recorded in its store in some specific order. For example, the partitionings could be ordered by increasing left partitions and the p/rhs pairs could be ordered first by right-hand sides and then by partitionings. For the purpose of this presentation, I consider that we want to output the parses in leftmost order. It is a simple matter to generalize the algorithm for other orders (see section 3.2.4).

The parse output process is as follows. For each right-hand side involved in the insertion of the distinguished symbol in the root node's set, the root node will send up all the parses relative to this right-hand side. For a given right-hand side, let us denote it $R_1 R_2$, for each partitioning of the input (pointer pair), the root will send up all the parses involving this right-hand side and this partitioning (p/rhs pair). To produce the parses related to a given p/rhs pair, the root proceeds as follows. To the partitioning correspond a left son (processor) and a right son. The root first asks the left son to send up the "first" sub-parse (of the sub-string it spans) relative to the non-terminal R_1 and asks the right son to send up its "first" sub-parse relative to the non-terminal R_2. (I shall explain later how these requests are transmitted from father to son). The root outputs the "first" parse relative to the current right-hand side ($R_1 R_2$) and the current partitioning by outputting its

own rule ($S \rightarrow R_1 R_2$) followed by the sub-parse provided by the left son followed by the one provided by the right son. To produce the "next" parse, the root asks the left son to send up again the "same" sub-parse it just sent while it asks the right son to send the "next" sub-parse relative to R_2. This is repeated until every combination of the first left sub-parse with each of the right sub-parses has been produced. The right son indicates to the root that it has sent its "last" sub-parse by attaching a *last-parse* marker to the last rule of its last sub-parse. The root takes this marker off before outputting the rule. The whole process is then repeated with the second (next) left sub-parse and then with the third and so on. Once every combination of left and right sub-parses (each time with the rule $S \rightarrow R_1 R_2$) has been output, the root switches to the next partitioning and outputs, using the same strategy, the parses relative to the new partitioning. It then switches to the next partitioning and then the next and so on. Once every partitioning associated with the first right-hand side involved in the reduction of the input string to the distinguished symbol has been considered, the root moves on to output the parses related to the second right-hand side and then to the third and so on. The very last rule that the root has to output has attached to it a *last-parse* marker. The root outputs this rule complete with its marker.

The processors send up the successive sub-parses of the sub-string they span relative to some non-terminal in the very same way that the root processor outputs the successive parses of the input relative to the distinguished symbol. In other words, the parse output process is recursive. The only difference between the root processor and the others is that the root will output in the process a given parse only once while the other processors may be asked to send up repeatedly the same sub-parse or the same sequence of sub-parses.

5.3.1 Requesting and providing sub-parses

When a father node requires a sub-parse from one of its sons it sends a request. Such a request can be transmitted straightforwardly by the link-nodes lying between the father and the son. There are three types of sub-parse request. I shall denote them by: FIRST-SP, SAME-SP and NEXT-SP. The sub-parses sought will always be those relative to the non-terminal labelling the tree-node. As we saw earlier on, the sub-parses are ordered. FIRST-SP requests that the processor sends the first sub-parse, SAME-SP requests that it sends the same sub-parse it sent last time while NEXT-SP asks for the following sub-parse. To be able to respond correctly to these requests, the processor must keep in its local memory a pointer to the p/rhs pair associated with the "current" sub-parse. Note that we will always want a processor that has just been marked as a tree-node to send the first sub-parse relative to its label. Hence, the marking of a processor as a tree-node implies a FIRST-SP request. The latter need not be sent explicitly. The sending up of the sub-parse as such can be realised via the same implementation as for the output of a single parse (see section 3.2.3).

5.3.2 Comments

The underlying parse trees

In the single parse algorithm, the marking phase reconfigures the array into some sort of tree of processors representing the parse tree of the input. The parse is then output. In the multiple parse algorithm the output of each parse involves a first phase similar to the single parse marking phase followed by an output phase which is exactly like the single parse output phase. In the multiple parse marking phase some tree-node processors send down tokens to mark their sons just as in the single parse marking phase but others only send down parse requests. As in the single

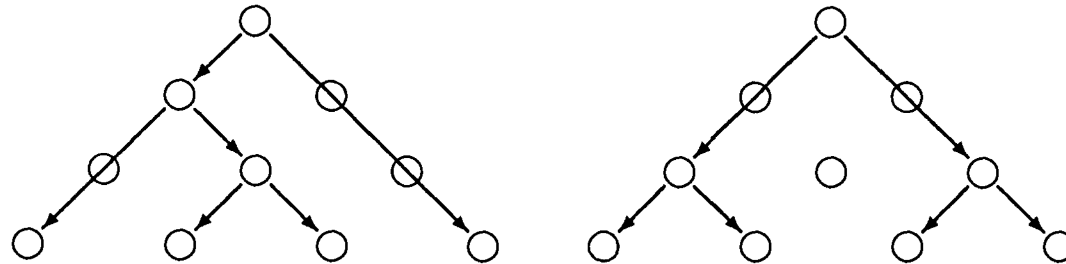
parse marking phase, the whole process is triggered by the root processor. Observe that if in the process a tree-node must mark its sons then all of its descendants (except for base processors) will also be required to mark their sons. For each parse output the parse tree represented in the array is modified. It successively takes on the shapes of all of the input's parse trees.

Impossible situations

A processor, either a tree-node or a link-node, that is sending up a sub-parse ought not to receive any request for another sub-parse before it has sent up the last rule of the current sub-parse. This is simply because such a request could only originate from the father of the node. But the father sends a request only after it has sent up its own sub-parse and thus only after its son has sent up the last rule of its sub-parse. If a processor has already received a FIRST-SP request, it ought never to receive another one before it has sent up the last rule of the last parse relative to its current label (but it may receive a SAME-SP request). This stems from the fact that if the father of the node requires any of its son's sub-parses (relative to a non-terminal) then it requires them all. For the same reason, a tree-node or a link-node processor can receive a marking token only after it has sent up the last rule of a last parse. A processor ought not to receive the sub-parse request NEXT-SP just after it has sent up the last rule of a last parse. This last rule will bear a *last-parse* marker and indicate to the father of the node that no next sub-parse exists and therefore the father shall not request one.

Lingering tree-node processors

A processor marked as a tree-node or a link-node may actually not be part of the current tree represented in the array. Consider for example the middle processor in the following arrays:



If the tree represented in the array changes from that one depicted on the left to that one depicted on the right then the middle processor will remain marked as a tree-node processor although it is not part of the second tree. This is because no marking token will have reached the processor. Such a lingering tree-node will not affect the extraction of the parse out of the second tree. The processor will receive no request at all during the extraction and will remain idle.

5.3.3 Complexity

Space

Assuming that we want our algorithm to be able to handle the worst cases and assuming that the processors of the array should all be identical, the analysis of the space complexity of our algorithm is as follows. Each processor has to have enough storage to hold a number of pointer pairs that is proportional to n where n is the length of the input, hence a number in $O(n)$. (In the very worst case, the hidden constant of this measure will be the number of right-hand sides of the grammar.) As in the single parse algorithm, the length of each pointer is $O(\log n)$. Hence, in each processor the storage requirement for the pointers is $O(n \log n)$. The remaining storage requirement of each processor (grammar, marks, tokens etc.) is fixed relative to the input size and hence the space complexity of each processor is $O(n \log n)$. The array consists of $n(n-1)/2$ processors. The overall space complexity of the algorithm is thus $O(n^3 \log n)$.

Time

The information processed by each processor in the recognition phase of the multiple parse algorithm is exactly the same as that processed in the recognition phase of the single parse algorithm. The information is also processed in the very same way except that in the former case the processors may record more of this information in their storage. The time required to store each item of the information is certainly independent of the input size. We may thus conclude that the recognition phases of both algorithms have the same time complexity, i.e., linear time complexity.

As I have mentioned earlier (section 5.3.2), the multiple parse algorithm does not have a separate marking phase. Instead, the marking of processors occurs at various points of time and the marking process is interleaved with the output process. However, if we look at what happens during the output of one parse in the multiple parse algorithm we can see that the output actually consists of two phases. I shall call them the *multiple parse marking phase* and the *multiple parse output phase.* The multiple parse marking phase is very much like the marking phase of the single parse algorithm while the multiple parse output phase is exactly the same as the output phase of this algorithm. What differs between the multiple parse and single parse marking phases is that in the latter a father will always mark its sons while in the former a father will on certain occasions mark its sons and on others it will simply send them a sub-parse request. Marking a processor that is at a distance d takes $2d$ beats while sending it a request takes d beats. Since the times of the processes differ by a constant factor we may conclude that the multiple parse marking phase (for a single parse) has the same time complexity as the single parse marking phase. We may thus conclude that the time taken by the multiple parse algorithm to output one parse is of the same complexity as the time taken by the single parse algorithm, i.e. linear in n.

The time for all the parses

To output one parse takes a time linear with the length of the input. Since the root, after it has output the last rule of a parse, takes a fixed amount of time to trigger the output of the next parse, the time to output all the parses is linear with the length of the input times the number of parses. The number of parses of a sentence depends of course on the specific grammar used and on the sentence itself. What can we expect in the worst case? It is easy to exhibit a grammar which can generate sentences whose numbers of parses are exponential functions of their length [Graham 76a].

A minute improvement

As I have mentioned two paragraphs back, it takes twice as long to mark a processor at some distance away as to send it a request. (Note that to send to a processor a request, it must be marked "properly" and so must the processors between it and the sender). It is thus preferable to have the fathers send requests to their sons as often as is possible and to limit the number of times they will need to mark their sons. In the description of the algorithm, I have suggested that the root outputs all the parses relative to a right-hand side first and then all the parses relative to another right-hand side and so on. The comment above indicates that it would be more efficient for the root to output all the parses relative to a given partitioning and then those relative to another partitioning and so on. This also applies to the order in which each processor sends up its sub-parse. Bringing this modification implies that each FIRST-SP request would have to carry a non-terminal to label the son.

This modification could only lower the time used to reconfigure the tree represented in the array. The factor it could lower it by is very much dependent on the particular input string. For example, suppose a string has 9 parses in all, that

there are 3 parses relative to each of 3 right-hand sides and that there are 3 parses relative to each of 3 partitionings. In this case, without the modification above the reconfigurations would imply 9 markings and no explicit request sending (a marking stands for an implicit request) while with the modification it would imply 3 markings and 6 explicit request sendings. Assuming a request transmission takes on average half the time of a marking, the modification would thus improve the reconfiguration time, in this instance, by a factor of roughly 2/3.

5.4 Recursive enumeration: the $O(n^2 \log n)$ method

In the space complexity of the multiple parse output algorithm described in the last section, a factor n is due to the fact that a string could be reduced to some right-hand side via all the partitionings of the string and the fact that the algorithm requires that processors keep in storage the information relative to every reduction made during the recognition phase. In the algorithm I am about to introduce this requirement is dropped. The result is an algorithm of space complexity $O(n^2 \log n)$. The basic idea is to have the processors record during the recognition phase enough information to output at least one parse but not necessarily enough to output all the parses. After the output of one parse, the recognition phase is rerun to allow the processors to "pick up" the information they may need to output the next parse. This process is repeated until all the parses have been output. The advantage we get from rerunning the recognition phase between successive parse outputs is that by doing so we limit the amount of information the processors may need to keep in their storage. The important fact is that the maximal quantity of information that can ever be required is independent of the input size. We do not rerun the recognition phase each time from scratch. The processors retain

throughout the rerun the information they used for the output of the previous parse. They use this information to select, among all the information they process during the recognition phase, what to record in preparation for the output of the next parse. This second multiple parse output algorithm is very similar to the first. Like the first, it resorts to FIRST-SP, SAME-SP and NEXT-SP requests and to last-parse markers. In fact about the only difference between the two algorithms is the repetition of the recognition phase. I assume, in the following, that the reader has well understood how the first algorithm works.

5.4.1 The information needed to output the next parse

Let us look back at what goes on at the level of a tree-node (or tree-node to be) processor during the output of one parse (with the first algorithm). During the output of the parse the processor must provide a sub-parse. Which specific sub-parse it must produce is determined basically by what it receives from above (either a marking token or some sub-parse request) and by some information held in its store (information related to reductions and the value of some state variables). I shall now address the following question: what is the minimal information the processor must have to be able to react properly to whatever request (a marking token bears an implicit request) it can receive? The string the processor spans could be reducible to any non-terminal of the grammar. It could be in fact reducible to every non-terminal. A string reducible to a non-terminal can be reducible to that non-terminal according to more than one right-hand side and for each such right-hand side it can be reducible according to more than one partitioning. Recall that the right-hand sides of the rules and the string partitionings are ordered in some way and that the processors are aware of this ordering.

If our processor receives one marking token (which marks it as a tree-node processor) carrying the value of some non-terminal, it must produce the first sub-

parse relative to this non-terminal (of the sub-string it spans). To be able to do so it will need to hold in its store the first p/rhs pair that brought the reduction of the processor's sub-string to this non-terminal. Since the value carried by the marking token could be any non-terminal, our processor must hold the first p/rhs pair associated with each non-terminal that was inserted in its set during the recognition phase. (In the worst case this set could equal the whole set of non-terminals of the grammar.)

If our processor receives a FIRST-SP request it must be the case that on some previous parse output it received a marking token carrying the value of some non-terminal. To honour the FIRST-SP request it only needs the first p/rhs pair associated with the non-terminal. It must also remember that it is a tree-node processor. (Only tree-node and link-node processors can receive sub-parse requests. The former must react to those requests while the latter must simply pass them on.) If our processor receives a SAME-SP request the matter is even simpler. In such a case the only information required by the processor is its status (mark) as a tree-node. For the processor to provide the sub-parse sought involves simply that it itself sends SAME-SP requests to its two sons. I assume that its sons have also kept their marks (tree-node) and that so have the processors leading to them (link-node). The last case, our processor receiving a NEXT-SP request, is a bit more complicated and also probably more interesting. The fact that the processor receives this request implies that it has already produced a sub-parse relative to some non-terminal (the one by which it is labelled) and that the next sub-parse relative to this non-terminal is sought. The last sub-parse sent up was relative to some p/rhs pair. Recall, there may be more than one sub-parse relative to the same pair. In the case that the next parse is relative to the same p/rhs pair, producing this sub-parse will imply that the processor simply sends the appropriate sub-parse requests (FIRST-SP, SAME-SP and NEXT-SP) to its sons. Which requests are appropriate depends entirely on whether the last sub-parse provided by the two sons carried with them last-parse markers

or not. For example, assuming the combination of left and right sub-parses are ordered as suggested in section 5.3, if the last left sub-parse did not have a last-parse marker and the right sub-parse did, the appropriate requests would be a NEXT-SP request for the left son and a FIRST-SP request for the right son. If it was the other way around, that is if the previous left sub-parse had a marker and the right one did not, then the processor would need to send its left son a SAME-SP request and its right son a NEXT-SP request. The important thing to notice here is that when the next sub-parse is relative to the same p/rhs pair, the (only) information needed by the processor is a record of whether the previous left and right sub-parses had last-parse markers or not. If the next sub-parse is relative to another p/rhs pair the processor will need to hold this pair. It will use this pair to mark its two sons. This second case occurs just when the previous left and right sub-parses passed up to the processor both carried *last-parse* markers.

I can now summarise what has been said above and indicate precisely what information processors require for the output of a next parse:

1. for each non-terminal in the processor's set, the first p/rhs pair involved in the insertion of the non-terminal in the set,
2. the most recent marking of the processor (link-node or tree-node),
3. the p/rhs pair that is next to the pair to which the last sub-parse produced by the processor was relative or an indication that the latter pair has no next pair,
4. a record of whether the sub-parses most recently passed up to the processor had *last-parse* markers or not.

Before explaining how we can arrange for the processors to hold the information they need when they need it, I shall add some further comments on this information.

First, the information enumerated above is not the minimum required at all times. For instance, if the last sub-parse produced by a processor was relative to some p/rhs pair and if this sub-parse was not the last one relative to this pair, the processor requires no other information apart from the indication of absence or presence of *last-parse* markers on the last sub-parses it received. That is because in such a circumstance the processor can only be requested to produce either the same sub-parse as the one produced last or the next one. Let us consider another case. Suppose a processor acted as a link-node during the output of the last parse. During the output of the next parse it is liable to get marked as a tree-node and labelled by any of the non-terminals in its set. To make sure it can cope with this eventuality we must have the processor record the first p/rhs pairs associated with those non-terminals. On the other hand, observe that our processor could get marked as a tree-node only if the last rule passed up during the output of the previous parse had a *last-parse* marker. If such is not the case, the only information it needs to retain is its link-node status.

Second, if a tree-node is outputting a sub-parse relative to the last p/rhs pair associated with a non-terminal, it must know that this pair is the last. In such a case if the sub-parses sent by its two sons both have *last-parse* markers the last rule sent up by the processor must also bear the *last-parse* marker. Many things can happen to a processor that has produced the last sub-parse relative to a non-terminal: it can be requested to produce the same parse again; it can be requested (explicitly) to produce the first parse relative to that non-terminal; it can get re-marked as a tree-node and labelled, either by that same non-terminal by which it was already labelled or by some other non-terminal of its set; it can get re-marked as a simple link-node; finally, it can just be left out altogether of the output process (not be part of the underlying parse tree(s) of the following parse(s)).

5.4.2 The recognition phases

Now that I have indicated what information is required for the output of a next parse, I shall explain how it is obtained. Recall that the idea of this algorithm is to have the processors keep in their storage as little information as possible but enough to output one parse and to rerun the recognition phase after each parse output so as to allow the processors to pick-up the information they may need for the output of the next parse. Actually, the recognition reruns are relevant to only one piece of the information listed in the last section. Their sole purpose is to allow (tree-node) processors to "pick-up" the right-hand side/partitioning pair that is next to the pair to which the last sub-parse they have produced was relative. The first p/rhs pair relative to each non-terminal of the processors' sets can be recorded during the first execution of the recognition phase and kept until the end. The indication of whether the last sub-parses passed up to a processor carried *last-parse* markers or not is of course obtained during the current parse output. This information is simply stored until needed in the next parse output. Finally, the link-node or tree-node status of a processor can also simply be kept in storage from one parse output to the next. A processor's status can be modified during the output of a parse.

I shall now show how a (tree-node) processor can get its "next pair" during a rerun of the recognition phase. For this, the processor must hold in its store what I call the "current p/rhs pair". This is the pair relative to which the sub-parse the processor produced during the previous output was. The only reason why the processor needs a record of this pair is so that it can spot the next pair, i.e. the p/rhs pair next to the current p/rhs pair. As I have mentioned previously, I assume that the right-hand sides of the grammar used and the string partitionings are ordered in some way. Which specific ordering is used is not important as long as the processors can establish which of two right-hand sides (or two partitionings)

comes first. To simplify the explanation however I shall adopt the following ordering of string partitionings.

A partitioning ordering

Say a string is of even length. Its first partitioning is the one that splits the string in the middle, the second splits it one position to the left of the middle, the third splits it one position to the right, the fourth two positions to the left and so on. By the same pattern, the first partitioning of an odd length string partitions it in a left sub-string that is one token shorter than the right sub-string. Vice versa for the second partitioning. The third partitioning involves a left sub-string two tokens shorter than the right sub-string and so on.

This ordering of string partitionings corresponds to the chronological order in which the processors receive , during the recognition phase, the values (non-terminal sets) computed by other processors. Let us consider for example a processor that is spanning an even length string. The first value pair it will receive will be from the two processors spanning the left half and right half of this string. These sub-strings correspond to the first partitioning of the string as defined by the ordering specified above. On the following beat our processor will receive two pairs of values corresponding to the second and third partitioning of the string. The second partitioning is the one that involves the shorter left sub-string. It corresponds to the pair of values arriving on the processor's left slow belt and right fast belt (see section 2.3.4). The third partitioning itself corresponds to the pair arriving (on the same beat) on the left fast belt and the right slow belt. From here on I shall denote the pair of belts consisting of the left slow belt and the right fast belt by the term *slow/fast belts* and likewise *fast/slow belts* shall mean the left fast belt and right slow belt pair. We can immediately see how a processor can tell whether the string partitioning corresponding to a value pair comes before or after the parti-

tioning corresponding to another pair. The partitioning corresponding to the pair that arrives first comes before the other. If both pairs arrive simultaneously, the partitioning corresponding to the pair arriving on the slow/fast belts comes first.

5.4.3 Recording the next pair

Let us now see how a processor can identify and pick-up its next p/rhs pair, assuming one exists. I shall use the term *current partitioning* to denote the partitioning of the current p/rhs pair. Likewise I shall use the terms *current rhs*, *next partitioning* and *next rhs*. For ease of presentation, I assume, without loss of generality, that the pairs are ordered first by partitionings and then by right-hand sides. During the recognition phase rerun the processor's counters will take on, and in the same order, exactly the same values they took on during the previous run. The processor will also receive exactly the same pairs of non-terminal sets that it received the last time. Because of the partitioning ordering we adopted, our processor will receive the pair of sets corresponding to the next partitioning pair (the one sought) either on the same beat or after the beat during which it will receive the pair of sets corresponding to the current partitioning pair (the one it already holds). I call the latter beat the *current partitioning beat*. The processor can easily identify this beat by comparing the values of its two counters with the values of the pair of pointers that represent the current partitioning. The beat is reached when both counters become equal to their corresponding pointers. (Since the sum of the two counters, and of the two pointers, is always the same, only one counter needs to be compared with its corresponding pointer). One of two things can happen. Either the next partitioning is the same as the current partitioning or it follows it. In the former case the next right-hand side must necessarily follow the current one while in the latter case the next right-hand side can be any right-hand side. Let us look at the first case first. On the current partitioning beat the processor receives a pair

of non-terminal sets which corresponds to the current partitioning. Because this partitioning is the same as the next one this pair of sets also corresponds to the next partitioning. It follows that the two non-terminals of the next right-hand side will be found in this pair of sets. Thus in this case, the processor, in its search for the next right-hand side, must look for the first right-hand side following the current one (in the ordering of right-hand sides) whose first non-terminal is in the first non-terminal set and whose second non-terminal is in the second set. The processor stores this right-hand side. It constitutes, together with the current partitioning, the next p/rhs pair sought. If no right-hand side meets the above conditions then it must be that the latter case prevails, i.e. the next partitioning follows the current partitioning. In this second case, the processor may receive the non-terminal set pair corresponding to the next partitioning either on the same beat or on a beat following the current partitioning beat. Recall that processors receive two pairs of non-terminal sets at a time and that these are ordered. In the ordering we have adopted, the pair arriving on the slow/fast belts comes before the other. Thus, if the pair of sets corresponding to the current partitioning arrives on the slow/fast belts, the pair corresponding to the next partitioning can arrive on the same beat on the fast/slow belts. Whatever the case may be, whether the set pair of the next partitioning arrives on the same beat or on a following beat, is not important. The important fact is that the processor can easily identify which of the pairs of sets it receives correspond to the partitionings that follow the current partitioning. These pairs are the ones that arrive on or after the current partitioning beat. The processor looks for the first pair amongst these that holds a rhs that reduces to the non-terminal by which the processor is labelled. This pair corresponds to the next partitioning sought. The processor must thus record the value of the pointers (counters) associated with it. The processor must also record the value of the rhs. If more than one rhs satisfies the above requirement, the processor picks the one

that comes first in the ordering of the right-hand sides. This rhs constitutes the next rhs. It and the next partitioning constitute the next p/rhs pair sought.

5.4.4 Identifying the last p/rhs pair

We have assumed above that a next p/rhs pair existed. But this may not be the case and it could be that the current p/rhs pair of the processor is the last one relative to the non-terminal currently labelling it. For the parse output algorithm to work, each tree-node processor must know whether the p/rhs pair relative to which it is producing a sub-parse is a last pair or not. Since a processor is liable to be labelled by any non-terminal of its set, it is liable to be required to produce a sub-parse relative to (the first p/rhs pair of) any non-terminal of its set. Thus, a processor must know whether each of the first p/rhs pair related to the non-terminals of its set is a last pair or not. Suppose a processor has just produced the last sub-parse relative to some p/rhs pair (relative to some non-terminal). This processor is then liable to produce either the same sub-parse or the first sub-parse relative to the same p/rhs pair or the first sub-parse relative to the next p/rhs pair (assuming there is a next p/rhs pair). To provide for the last possibility, the processor must thus know whether the next p/rhs pair is a last pair or not.

How can a processor obtain this information? We have seen in the last section how a processor that holds some p/rhs pair related to some non-terminal can pick-up the next pair during the recognition rerun. If no such pair can be found it is simply that the pair already held is the last p/rhs pair relative to the non-terminal. Thus, to find out if the first p/rhs pair relative to some non-terminal is also the last, a processor can proceed as follows. Once it has recorded the first pair (i.e. once it has inserted the non-terminal in its set), the processor looks for a next pair using the method described above. If and only if the processor cannot find a next pair relative to the same non-terminal then the first pair is also the last. If the

processor finds a next pair, it does not need to store any information relative to it. It just needs to record the fact that the first p/rhs pair is not a last pair. Each processor does this for each non-terminal that it inserts in its set. Observe that finding the first p/rhs pairs and finding out whether they are also last pairs need to be done only once (during the first execution of the recognition phase).

The purpose of the recognition phase rerun is to allow the processors to pick up a p/rhs pair that is next to some pair it already holds (its current pair). Using the same strategy as above, the processor can determine if the next pair picked up is a last pair simply by searching, after it has found the next pair sought, for another pair that follows it in the ordering.

5.4.5 The next becomes the current

Recall that the sub-parse produced by the processor during the last parse output is relative to the pair that has been referred to in the last section as the current p/rhs pair. During the following parse output, the processor may produce a sub-parse relative to that same pair or it may also be requested to produce a sub-parse relative to the next p/rhs pair (there are also other possible outcomes). In the former case, the current pair remains the current pair while in the latter, the next pair becomes the current pair. In this last instance, the processor no longer needs to retain the information relative to the old current p/rhs pair. If the processor is ever required again in the future to output a sub-parse relative to that pair, it will have picked it up first during a preceding rerun of the recognition phase.

5.4.6 Alternative orderings

The ordering of partitionings that I suggested earlier makes it easy for the processors to spot their next p/rhs pair. The processors first wait until they receive the pair of

non-terminal sets corresponding to their current partitioning and from then on they look for a next pair. That ordering corresponds to the chronological order in which processors receive non-terminal sets from other processors. It is not a very natural ordering. A more natural way to order the partitionings would be, for example, to order them by increasing left partitions. I shall show how the algorithm presented above can be adapted for this alternate ordering. Earlier, I have also assumed that the p/rhs pairs are ordered first by partitionings and then by rhs. I shall also show how the algorithm can be adapted for the case where it is required that the p/rhs pairs should be ordered by right-hand sides first. From these two examples, the reader should get the general idea of how the algorithm could be adapted for various other p/rhs pair orderings.

Order of increasing left partitions

Recall that during the recognition phase, a processor receives from its immediate lower left neighbour the sets of non-terminals that this neighbour and the ones further along in the lower left direction have computed. For ease of explanation, I shall assume that our processor is at an odd distance from the base (i.e. that it has an odd number of lower left neighbours). The first set received is from the processor that is halfway between our processor and the base. (Our processor gets two copies of this set. One comes on the slow belt while the other comes on the fast belt.) The set received corresponds to the left half (partition) of the string spanned by the processor. On the following beat, the processor receives on its slow belt a set coming from the processor that is one position further than halfway and it receives on its fast belt a set from a processor that is one position closer. These correspond to left partitions that are respectively one token shorter and one token longer than the previous (half length) partition. On the next beat, the processor receives sets corresponding to left partitions that are respectively one token shorter

and one token longer than the previous ones and so on. In other words, the sets received by the processor on its slow belt arrive in the order of decreasing size of the partitions to which they correspond (starting from the half length partition) while those received on its fast belt arrive in increasing order of corresponding partitions. In the case where the set corresponding to the left partition of the current p/rhs pair (let us call this set the *current left set*) arrives on the fast left belt, the problem of spotting the next p/rhs pair is similar to what it was before. It is in the other case where things become slightly more interesting. If the processor is to receive the current left set on the slow belt then it may receive the left set corresponding to the left partition of the next p/rhs pair (let us call it the *next left set*) before the current left set. If this happens then we must have the processor record the next pair before it receives the current one. The strategy to achieve this is fairly simple. If the processor receives a pair which could be a next pair it records it as a candidate. If later on the processor receives another pair which precedes the one recorded (in the ordering) and if it follows the current pair then this pair replaces the one recorded as a candidate. The details are as follows. During the recognition rerun, the processor records the first p/rhs pair (relative to its label) it can deduce whether the associated sets arrived on the slow/fast belts or the fast/slow belts. If on the same beat both the set pair arriving on the slow/fast belts and the one arriving on the fast/slow belts allow the processor to deduce a p/rhs pair, the processor records the one deduced from the former set pair. This is because that pair involves a smaller left partition than the latter. If a set pair allows for the deduction of more than one p/rhs pair, the processor favors the one which contains the rhs that comes before the others in the rhs ordering. Then and until it receives the current p/rhs pair, the processor does the following. If it receives on the slow/fast belts a set pair which allows it to deduce another p/rhs pair (always favoring the lowest ordered rhs if more than one can be deduced), it replaces the pair recorded by that other pair since the latter involves a smaller left

partition. When the processor receives the sets corresponding to the current pair it may be the case that the processor does not have in its store a candidate. If it has a candidate and if the set pair of the current p/rhs pair arrived on the slow/fast belts then the candidate is a good one. It gets nominated the next pair and we are through. If on the other hand the set pair of the current pair arrived on the fast/slow belts then its left partition would be longer than that of any candidate the processor could have recorded and so such a candidate would have to be rejected. In that case or in the case where no candidate has been recorded the processor must keep looking for the next pair. It looks first in the set pair corresponding to the current partitioning. In that pair, it looks only for right-hand sides that follow the current one. If it cannot find one in that set pair it shall look for any rhs in the set pairs that it will receive on its fast/slow belts. If the set pair corresponding to the current p/rhs pair arrived on the slow/fast belts, it looks first for a rhs in the set pair that arrived on the same beat on the fast/slow belts. From then on the processor must ignore the set pairs arriving on the slow/fast belt since these involve left partitions that are smaller than the current left partition.

Ordering first by rhs

For this explanation, let us revert back to the partitioning ordering that was suggested originally. If the p/rhs pairs are ordered by rhs first and then by partitionings then the pair next to some current pair could consist of the same rhs as the current rhs and a partitioning that follows the current partitioning. If such a pair does not exist then the next pair could consist of some rhs that follows the current rhs and of any partitioning. If more than one such pair exist, the next one is the one with the lowest rhs among them and if more than one such pair exist, the next one is the one with the lowest partitioning. To implement the search of the next pair with this ordering, we apply the same strategy as above. That is as soon as a (tree-node)

processor starts receiving set pairs, it looks for a candidate next p/rhs pair. Until it receives the set pair of the current partitioning, the processor records any p/rhs pair it can deduce that has a rhs which follows the current one. It replaces the recorded pair if it finds a pair with a rhs that follows the current rhs and precedes the recorded rhs. That is because in the overall ordering, any p/rhs pair with a lower order rhs precedes any pair with a higher order rhs. Once it has received the set pair of the current partitioning, the processor keeps doing the same thing except that from this point, it can also record a p/rhs pair containing the same rhs as the current pair. That is because from then on, the partitionings involved follow in the ordering the current partitioning. If the processor finds such a pair then it has found the next pair and it stops searching. Otherwise, the processor keeps looking for a better pair up until it receives the last set pair. At that point, if no definite next pair has been found, the recorded candidate pair, if one exists, gets elected to the post. If none exists, then it is the case that the current p/rhs pair was the last pair.

It is a simple matter to combine the ideas presented in the last two sections if we want the p/rhs pairs to be ordered by rhs first and if we want the partitionings to be ordered by increasing left partitions.

5.4.7 Complexity

Space

As was mentioned at the beginning of the description of this second algorithm, the main purpose for introducing the recognition phase reruns was to limit the number of information elements that a processor could be required to hold to a number independent of the input (or array) size. As was shown in section 5.4.1, the maximal information that the processors need to hold for the algorithm to work properly

amounts to one p/rhs pair for each non-terminal in the grammar and possibly one more pair plus a few status flags. The number of pairs is thus independent of the input size. The space required to store one such pair, because it consists of pointers, is $O(\log n)$ (where n is the input size). Thus, the space requirement for one processor is $O(\log n)$ and thus for the whole array, it is $O(n^2 \log n)$.

Time

The output of each parse involves a recognition phase and a marking and output phase. The marking and output phases are exactly the same as the corresponding phases of the previous algorithm and so they are of linear time complexity. The recognition phase of the second algorithm however is slightly more elaborate than that of the first. All of the extra operations that must be performed during each beat of the phase, except for the comparison of pointers (to detect the arrival of the current p/rhs pair) certainly take fixed amounts of time. If we want to be totally rigorous we have to take into account the fact that comparing two values of $\log n$ bits takes a time $O(\log n)$. We have a way out however. Recall that the counter implementation described in section 3.4 allows a *test if zero* (IF0) operation to be performed in constant time. We can implement the detection of the current p/rhs pair arrival as follows. When a processor starts to receive set pairs (this is indicated in GKT by a control signal), it resets a special counter to zero which it increments it on each of the following beats. Whenever the processor records a p/rhs pair (it could be a first pair or a candidate for a next pair or whatever...) it records, with the pair, the value of this counter. If on a subsequent recognition phase, if the processor needs to identify the beat on which the set pair corresponding to this p/rhs pair arrives, it does this. On the beat it receives the first set pairs, it initialises a counter to the value mentioned above (the value the special counter had when the p/rhs pair was recorded during a previous phase). It decrements this

counter on each following beat. When the counter reaches zero, the current beat is the one we wanted to detect. Assuming the above implementation for the detection of the current p/rhs pair arrival, we conclude that the recognition phase also has a linear time complexity. We thus conclude that the time to output each parse with the second algorithm is as with the first $O(n)$.

A net gain

We have an interesting result here. By repeating the recognition phase between parse outputs we have reduced the space complexity of multiple parse output by a factor of n. While in doing so, we have certainly increased the time taken to output the parses, but we have increased it only by a constant factor.

Chapter 6

Problem decomposition

6.1 Introduction

Up until now I have assumed that we always had at our disposal an array of processors as large as the size of the input required it to be. Considering that for an input of size n, the number of processors needed is $n(n+1)/2$, this assumption may not be very realistic, mostly in the specific case of CFL parsing. In this chapter, I address the issue of how we can solve, given an array of some fixed size, problem instances too large for the array. I restrict myself and consider only that part of the problem dealing with the computation of the recognition (solution) matrix, i.e. the recognition phase. The existence of a simple efficient sequential algorithm for the extraction of the parse out of the recognition matrix makes the use of a processor array unattractive in the circumstances.

I resort to a standard technique which handles a large instance by simulating a *virtual* array on a much smaller *real* array. The technique resorts to auxiliary storage that must be accessible to the processors at the boundary of the real array.

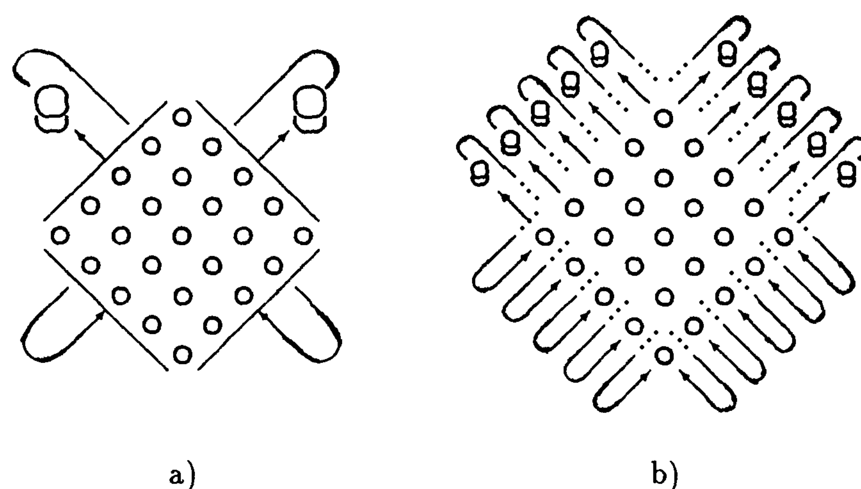


Figure 6–1: The real array.

I first describe the technique briefly in the context of the recognition phase. I then address various issues related to the use of this technique. More specifically, I look at the question of auxiliary storage space, timing, counter initialisation and recognition matrix recovery. Finally, I present an analysis of the space and time complexity of the algorithm.

6.1.1 The real array

The array at our disposal is an $s \times s$ rectangular array of processors for some $s > 0$. Auxiliary storage devices are connected to the array. The top left and the top right processors of the array can write on auxiliary store while the bottom processors can read data from it. In the algorithm, a bottom processor needs to read from the store only data that have been written by the processor in line with it on the opposite boundary of the array. Hence, the auxiliary storage devices could either consist of two devices, one for each pair of opposite array boundaries (see figure 6–1a), or of 2s devices, one for each pair of opposite boundary processors (see figure 6–1b). I assume that the main mode of data access of the storage devices is sequential and

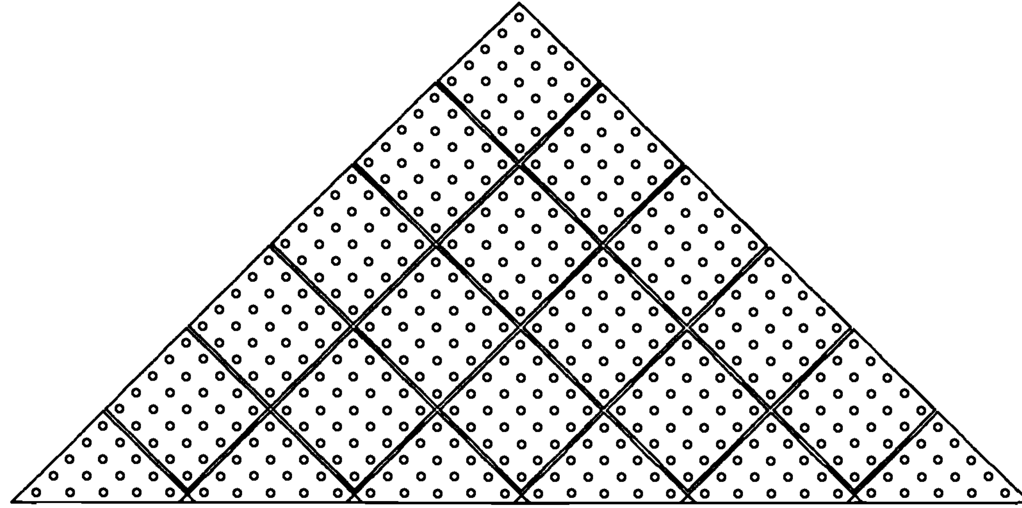


Figure 6–2: The virtual array.

that the rate of transmission is in accordance with the rate at which the processors of the array exchange data. The devices also have the capability to start reading or writing data from specified locations in the auxiliary storage.

6.1.2 The virtual array

The algorithm is based on the concept of a virtual array. Suppose the size of the input is n. For sake of simplicity I assume that n is a multiple of s, i.e. $n = qs$ for some integer $q > 1$. If we had an $n \times n$ array (depicted in figure 6–2, in this figure $q = 5$) we could then use the original version of the algorithm. But our array is much smaller. I have drawn sub-divisions in the figure of the virtual $n \times n$ array. I suppose each diamond shape sub-division is of the size of our real array and that each triangular sub-division at the base is (slightly over) half the size of our array (the upper half). I shall call diamond shape sub-divisions *full-tiles* and triangular ones *half-tiles*. I shall also use the word *tile* to refer to either full-tiles or half-tiles.

Bottom processors of a full-tile receive data only from top processors of adjoining full-tiles or half-tiles. The idea of the algorithm is to simulate the virtual array using the real array by multiplexing in time the processing that should be executed by each tile. We first simulate the bottom leftmost half-tile by running the original algorithm on the top half of the real array, feeding it as input the first s tokens of the input string. The recognition phase proceeds as usual except that some of the information coming out of the top right processors gets stored in auxiliary storage (instead of being left to wander in ether!). After the topmost processor has computed its value, only null values flow out of the processors on the top right boundary. At that point, we stop the execution of the algorithm, reset the array and rerun the algorithm with the next s tokens as input. This time, information emerging from both the top left and top right boundary processors gets saved. We continue in this fashion until we have simulated the processing of each half-tile at the base of the virtual array. We then simulate the processing of the full-tiles above. We simulate a full-tile by running the original algorithm on the whole array and by feeding the bottom processors of the array with the relevant information that was saved during simulation of the adjoining tiles below. This is the basic idea. I now examine in more details some of its implications.

One can resort to other strategies than the one presented above to have a small real array simulate a large virtual array. One of these, for example, consists in having each processor simulate a square array of virtual processors. Such a strategy can be applied if the real processors are relatively powerful. I shall have more to say about this in the next chapter in which I take a look at actual real machines.

6.2 The information in auxiliary storage

The information we need to store in auxiliary storage are the processor values emerging out of the fast and slow belts of the topmost processors of the real array. A topmost processor of the real array always plays the role of some (virtual) processor in the virtual array that is located on the top boundary of a tile. The information saved is the one this virtual processor would transmit to its virtual neighbour above located at the bottom boundary of an adjoining full-tile. We save the information so as to be able to feed it later on to the real bottom processor opposite the topmost processor when the former will play the role of the virtual neighbour.

6.2.1 The amount of information

Not all that is coming out of a topmost processor's fast and slow belts is worth saving. Firstly, at the beginning of the algorithm's execution only null values emerge out of processors that are high up in the array. (The higher up a processor is in the array, the longer it will take before it outputs non-null values.) Also, once a processor has computed and sent up its own value, it receives and transmits only null values. Clearly, there is no point in storing null values in auxiliary storage if we can help it. Secondly, in the GKT version of the basic algorithm, a processor may put on its fast belt a non-null value and on the same beat put a null value on its slow belt. The non-null value travelling on the fast belt will never be caught up by any non-null value on the slow belt and it will never be involved in the computation of any of the other processors' sets. Here again, we shall not clutter the auxiliary storage needlessly with such values. (In the Kosaraju version of the basic algorithm no non-null value travels on a fast belt accompanied by null values on the slow belt. It is very easy to modify the GKT version so that it behaves in the same way.)

Recall that a processor at a distance d from the base computes its value from beat $2d/3$ to beat $2d-1$. During that period, the processor receives on its left slow belts the values that have been computed by the processors on the same row lying between the base and halfway between the base and the processor. On its fast belt it receives the values of the processors lying halfway between it and the base. Now, suppose that our processor is a virtual processor located on the bottom left boundary of a full-tile. Then, the values mentioned above are precisely those that the real processor will need to obtain from the auxiliary storage when it will play the role of this virtual processor. Of course, these are also the values that the neighbouring virtual processor located on the top right boundary of the bottom left adjoining tile will have sent out on its top right fast and slow belts. The values will have been provided at some earlier time by the real processor playing the role of this virtual processor. We will have saved these values in auxiliary storage at that time. The comment at the beginning of this paragraph implies that the number of values that needs to be saved is equal to d, the distance of the virtual processor from the base. Half of these values will be for feeding into the processor's fast belt and the other half for feeding into its slow belt.

6.2.2 The number of values produced by a row of a half-tile

Say a row of a half-tile consists of r processors ($1 \leq r \leq s$). During the simulation of this half-tile the corresponding r processors of the real array will compute r processor values (sets of non-terminals). These values will all start their journey upward on the fast belt. Only half of those however will exit out of the row's topmost processor's fast belt. These $r/2$ values will be from the processors of the top half of the row. The other half of the values computed will have been transferred onto the slow belt and they will exit out of the row's topmost processor's slow belt.

These $r/2$ values will be from processors on the bottom half of the row. I assume here that as in the Kosaraju version of the basic algorithm, no copy of a value that has been transferred from a fast to a slow belt remains on the fast belt. The total number of values the row will produce is thus r.

6.2.3 The number of values added by a row of a full-tile

Say a virtual processor at a bottom left boundary of a full-tile is at a distance d from the base, $d > s$. When the real processor located at the corresponding position on the bottom left boundary of the real array will simulate this virtual processor, it will receive $d/2$ values on the fast belt and the same amount on the slow belt from the auxiliary storage. During the simulation of the full-tile, the s processors of the row on which the real processor lies will compute s values. The first of these will be computed by the bottommost processor of the row. It will have been computed only after the processor will have received all of the d values from auxiliary storage. Of the $d/2$ values received on the fast belt, $s/2$ will be transferred onto the slow belt and only the rest will reach the top end of the fast belt. Because $d > s$, all of the s newly computed values will also reach the fast belt top end. Hence, $(d+s)/2$ values will exit out of the row's top processor's fast belt. These values will be saved for later. All of the $d/2$ values fed in the lower end of the row's slow belt will reach the top end and so will the $s/2$ values that will have been transferred from the fast belt. Hence, the number of values reaching the top end of the slow belt will be the same as the number of values reaching the end of the fast belt. The total number of values saved will be $d+s$. The row will thus have produced s new values. Observe that these values are intended for a virtual processor that is at a distance $d+s$ from the base.

6.3 Timing issues

The idea of simulating the virtual array with the real array is fairly simple. If a real processor does not have a top (left or right) neighbour but the corresponding virtual processor has one then we save the values the real processor outputs so as to be able to feed them later to a real processor that will stand for the virtual top neighbour. We have just seen what values need to be saved and how many need to be saved. I now consider the question of timing.

Let us look at the values coming out of the top right boundary of the real array while it is simulating a half-tile. I assume that at time 0 the relevant portion of the input is available to the processors of the real array corresponding to base processors of the virtual array. As in the Kosaraju version of the basic algorithm, processors never send a non-null value on their fast belt with a null value on their slow belt. On beat 1 the bottom rightmost processor on the top right boundary outputs its own value. On the same beat all the other processors on that boundary output only null values. On the following beat the processors all output null values. On beat 2 the bottom rightmost processor outputs only null values, its top left neighbour outputs non-null values, all the other processors output only null values. On beat 4 and 5 the third processor from the bottom right outputs non-null values while all the others output only null values and so on. On beat $\lceil 3s/2 \rceil$ the topmost processor outputs its first non-null value and on beat $2s$ the last non-null value that is to come out of the array comes out of the topmost processor. Figure 6–3 shows schematically what goes on. On that figure circles ($\circ$) represent active processors, diamonds ($\diamond$) represent idle processors, bullets ($\bullet$) represent pairs of non-null values and dots ($\cdot$) represent pairs of null values. The values depicted are the ones we will want to feed in on the other side of the array when we make it simulate the full-tile above and to the right of the current half-tile. Notice that a processor on the top

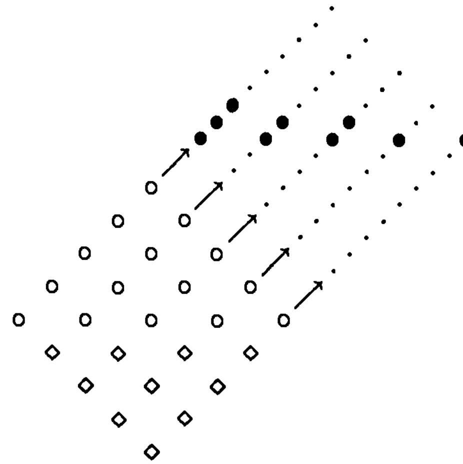


Figure 6–3: The output from a half-tile.

right boundary at a distance d from the (virtual) base will output $\lceil 3d/2 \rceil$ leading pairs of null values before outputting its first pair of non-null values. The simplest thing to do is to store these leading null value pairs in auxiliary storage as they come out. Alternatively, we can decide that we want to store only non-null values. This implies that we insert appropriate delays between the start of the simulation of a full-tile and the times when we start feeding in streams of values from the auxiliary storage into each of the processors at the bottom boundary of the array. Such delays could be implemented easily with two control signals that would leave the bottom corner of the array and travel at the speed of $2/3$ processor per beat along each bottom boundary.

6.3.1 Doing only useful work

Let us now look at the values coming out of the top-right boundary of a full-tile. Let us consider a full-tile at the lowest level in the virtual array. Figure 6–4 depicts the data that are fed in the bottom-left boundary of the real array and the data

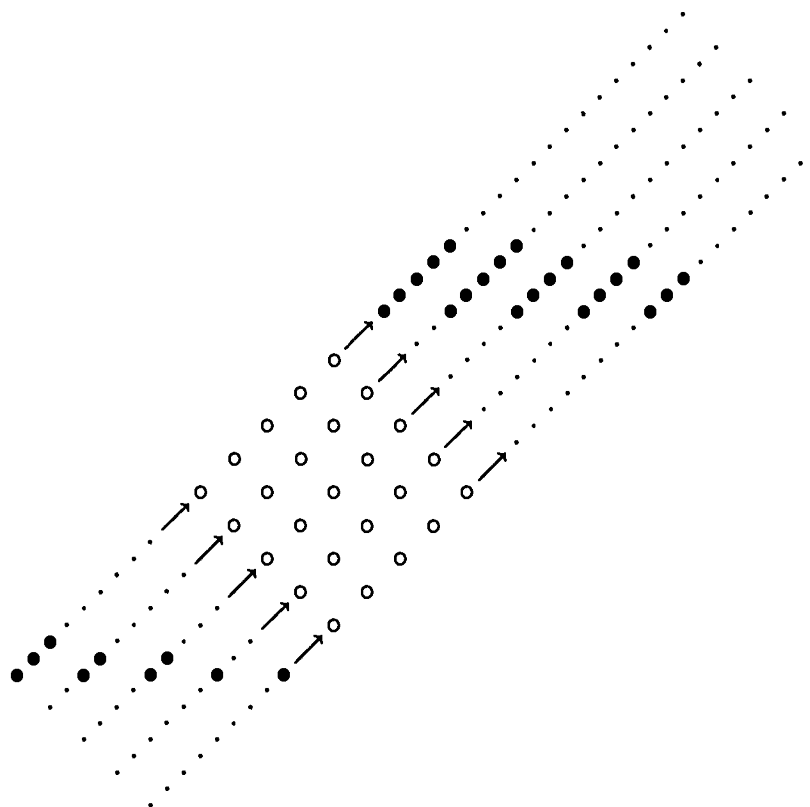


Figure 6–4: The output from a full-tile.

that come out of the top-right boundary. Notice that 8 beats elapse before the bottom rightmost processor of the top-right boundary outputs its first non-null value. During these 8 beats all of the processors on that boundary output only null values. It would be both a waste of space and a waste of time to store these values in auxiliary storage. We cannot do otherwise but wait for the 8 beats to elapse before we start collecting useful values coming out of the array. But if we do not store the null values produced during the initial 8 beats period and hence if we do not feed them back in the bottom-left boundary when the time comes to simulate the tile above and to the right of the current tile, then we will be able to avoid an unnecessary 8 beat delay. Doing so amounts to simulating the processing of this

second tile only when it is doing useful work, i.e. when its processors are not all just exchanging null values. Since a processor at a distance d from the base computes its value from beat $\lfloor 3d/2 \rfloor$ to beat $2d - 1$, simulating the virtual tiles only when they are doing useful work should take roughly $1/4$ of the time it would take to simulate them from the start of the virtual algorithm execution.

6.3.2 The fast belt and slow belt paradox

When we have the array simulate a full-tile we start feeding (non-null) values onto the fast belt of the bottom processor of the row at the same time that we start feeding (non-null) values onto its slow belt. Paradoxically, although the slow belt carries values at half the speed of the fast belt, non-null values start coming out of the fast and slow belts of the top processor of the row at the same time. This is explained by the fact that some values on the fast belt get transferred onto the slow belt. In fact, as soon as a non-null value on the fast belt passes in front of the stream of non-null values on the slow belt the basic K-GKT algorithm has this non-null value move onto the slow belt. Hence, the fronts of the two streams of non-null values are never further than one value apart from each other.

As many values are fed onto the fast belt of the bottom processor of the row as onto its slow belt. Although, as we have just mentioned, some of the values initially on the fast belt end-up on the slow belt, as many values come out of the top processor's fast belt as out of its slow belt. This is explained by the fact that each processor of the row computes a new value and deposits it onto the fast belt. The number of new values is double the number of values transferred from the fast belt to the slow belt.

6.4 Initialisation of counters

Up until now I have dealt with matters that concern only the basic K-GKT algorithm. The one question I have to consider with regard to my extension to this algorithm is the question of initialisation of counters. In section 3.2.1 I suggested two methods for the initialisation of counters. I show how these methods can be adapted for the execution of the algorithm on the virtual array. When the real array simulates a half-tile, the original methods can be applied without modification. I shall thus consider only full-tiles.

Recall that when a processor at a distance d from the base starts to compute its value, its two counters must be initialised to $\lceil d/2 \rceil$ and $\lfloor d/2 \rfloor + 1$ respectively. For a real processor $P_{i,j}$ let me define its *distance from the bottom corner* of the $(s \times s)$ real array, which I shall denote d', as the value $(j-1)+(s-i)$. (By the convention I have been using, $P_{1,1}$ denotes the left corner processor of the array while $P_{s,1}$ denotes the bottom corner processor.) The distance of a processor from the bottom corner is equal to the distance between this processor and the processor on the same row on the bottom-left boundary of the array plus the distance between the latter processor and the bottom corner processor. The distance between the base and the virtual processor a real processor stands for (during the simulation of a full-tile) is equal to the distance between this real processor and the bottom corner processor plus the distance between the base and the virtual processor the bottom corner processor stands for. To implement the initialisations of the counters we precompute, before the simulation of a full-tile, the half of the distance from the base of the virtual processor located at the bottom corner of the full-tile. I call this distance the *bottom corner offset.* At the start of the simulation we transmit the bottom corner offset to the bottom corner processor of the real array. We broadcast this value to all the processors of the array by having the bottom corner processor

send it to its two upper neighbours and by having any processor that receives the value on a beat send it to both of its upper neighbours on the next beat. Using either of the two counter initialisation methods suggested in chapter 3 we have each processor increment their counters to the values $\lceil d'/2 \rceil$ and $\lfloor d'/2 \rfloor + 1$ where d' is the distance between a processor and the bottom corner. The original methods use control signals that leave at the base of the array and travel either along its rows or its columns. To achieve the required result here, we use the same signals but we have them start off from the bottom corner of the array and have them travel along both the bottom row and the bottom column. On each beat, each signal gets duplicated. A processor that has received a signal, either from its bottom left or bottom right neighbour, will send a copy of it to both of its upper neighbours. (We proceed in this way just to ensure that every processor of the array receives the signals. We could also have only the processors on either the bottom row or bottom column duplicate the control signals.) On beat $\lfloor 3d'/2 \rfloor$ the counters of a processor at a distance d' from the bottom corner will have reached the values $\lceil d'/2 \rceil$ and $\lfloor d'/2 \rfloor + 1$. At that time, the processor will have already received the offset value. It will add this offset value to each of its counter values to obtain the final values required.

Let us assume that s, the number of processors on a side of the array, is even. In that case the distance between the bottom corner processor of any full-tile of the virtual array and the base of the virtual array will always be odd. Our counters are integers so the offset value will have to be an integer. We may choose it to be either the floor or the ceiling of the corresponding real value. Whatever choice we make is not important as long as we adjust the counter initialisation appropriately. The necessary adjustment must ensure that the final values of the counter are not off by one from the required values. The details are not interesting and I skip them.

As a final comment on counter initialisation, observe that the computation of the counter values before the addition of the offset need not to be done more than

once. Actually, if we were to build an array, we would probably prefer to store these values permanently in each processor rather than to implement the whole counter initialisation method described in chapter 3.

6.5 Recovering the recognition (solution) matrix

In order to be able to produce the parse of the input (using the usual sequential method [Aho 72]) we have to recover the recognition matrix that will have been computed by the virtual array. Each element of the matrix must consist not only of a set of non-terminals but also of the associated rules (rhs) and pointer pairs that will have been involved in the insertions of the non-terminals in the set. The recovery of the recognition matrix can be achieved in various ways. One simple method consist in collecting its elements as they come out of the top-right (or top-left) boundary of the real array while it is simulating each of the tiles located on the top-right (or top-left) boundary of the virtual array. Recall that the non-null values a (virtual) processor receives on its fast belt from its bottom-left (bottom-right) neighbour are the values that have been computed by the (virtual) processors located along the same row (column) between it and halfway between it and the base. It receives on its corresponding slow belt the other values that have been computed on its row (column). Thus, when a real processor simulates a top-right (top-left) virtual boundary processor, it outputs on its top-right (top-left) fast belt the values of the top half of the row (column) of this virtual processor and it outputs on its slow belt the values of the other half. These values are just those we are seeking (from this row of the matrix). Recall that the values come out of the fast belt in the reverse order in which they come out of the slow belt. If the values

are stored in a random-access memory, it is a simple matter to store them in some given order.

The major drawback of the method presented above is that it implies that throughout the simulation of the virtual array, the values exchanged by the (real) processors and stored in auxiliary storage will have to consist not only of non-terminal sets but also of rules and pointer pairs. It will thus take more time to perform the exchanges of processor values and these will occupy more auxiliary storage space. Notice that for the computation of its value a processor requires only the non-terminal sets. The rules and pointer pairs associated with these are needed only after the recognition phase. To recover the recognition matrix while avoiding the unnecessary movement of rules and pointer pairs we can proceed as follows. Once the real array has terminated the simulation of a tile, we have the array go through what I call the *flushing* phase. During this phase, we extract the values (non-terminal sets, rules and pointer pairs) that have been computed by the array during the simulation. This can be achieved by first having each processor deposit its value on one of its fast belts, say the row oriented one, and then having all the values migrate out of the array through, say, the top-right boundary. In this fashion, we can recover the whole matrix by recovering each of its tiles separately.

We can combine the two previous methods and obtain yet a slightly faster method. We can use the flushing phases to recover only the rule and pointer pair components of the recognition matrix and use the first method to recover the non-terminal set component. Since in that way the processors would move less information during each flushing phase that phase could run more quickly. And since when the real array simulates a top-right boundary tile the non-terminal sets sought flow out of the real array anyway, no time need be lost in recovering these values as they come out.

6.6 Complexity

6.6.1 Space

Since our real array is of some fixed size, we may consider its number of pointers to be a constant. Since the size of the input we can handle with our array is limited by the size of the counters in the processors, the space complexity due to the array itself is $O(\log n)$ where n is the size of the input. In order to evaluate the space complexity of the algorithm we must also take into account the amount of auxiliary storage required. We have seen in section 6.2.1 that the amount of information we must save for the simulation of a tile is proportional to the height of the tile in the array. After having simulated all the half-tiles at the base of the virtual array we will have in auxiliary storage an amount of information proportional to n since the number of half-tiles simulated is proportional to n (it is equal to n/s) and each simulation will have produced an amount of information proportional to the size of the real array, which is constant. Just before simulating the full-tile at the top of the array we shall also have in storage an amount of information proportional to n. This is simply because the top tile is at a distance from the base proportional to n. When we have simulated or just before we simulate the full-tiles that are located mid-way between the base and the top of the virtual array we shall have in storage an amount of information proportional to n^2. This follows from the fact that the distance between these tiles and the base is proportional to n and that the number of such tiles is also proportional to n (is equal to $n/2s$). I show that the maximal amount of information that we may ever need to store cannot be greater than some constant factor times n^2. Observe that the information needed to simulate a tile of the virtual array (or, should I say, to compute a tile of the recognition matrix) comes from a portion of the recognition matrix. The information required is the set

of values of the recognition matrix elements in the tiles below and on the same row (forward diagonal) as the tile we want to compute (simulate) and those in the tiles below and on the same column (backward diagonal). Since we can simulate a tile only after having simulated all the tiles below in the same row (column) and since we must simulate it before we simulate any tile above in the same row (column) it follows that we shall not need to have in storage more than two copies of any tile of the recognition matrix. We might have a copy for the simulation of at most one tile above in the same row and another for the simulation of at most one tile above in the same column. Since the number of values in the whole recognition matrix is $O(n^2)$ it follows that the maximal number of values we may need to store is $O(n^2)$. Because the values must include counters at one point or another, we must add a factor of $\log n$ to this measure. Hence the space complexity of the whole algorithm (recognition phase) is $O(n^2 \log n)$. We obtain the same result as when we considered the array always to be large enough for our inputs. This is explained by the fact that storing the information in an array of processors or storing it in some auxiliary storage device does not affect the amount of information that needs to be stored. The space complexity is also the same as in the sequential method (taking pointers into account).

6.6.2 Time

Let us determine the total number of beats required to simulate the whole virtual array for an input of length n using an $s \times s$ real array. I assume that $n = qs$ for some integer q. I define virtual array *tile-levels* (which are similar to real array levels (section 3.1.1)). The tile at the top of the virtual array is at tile-level 1. The 2 tiles below are at tile-level 2 and so on. The half-tiles at the base are at tile-level q. The simulation of each half-tile requires the same number of beats as the execution of the original algorithm on an input of s tokens, i.e. $2s$ beats. There are q half-tiles

and so the simulation of all the half-tiles requires $2qs$ beats. As mentioned above, we only simulate the useful processing of a tile. Hence, the amount of time required for the simulation of a tile is the same as the amount of time during which such a tile would be doing useful work if it was actually part of a real array as large as the virtual array. Let us consider a tile at tile-level l, $l > 1$. Its bottom corner processor is the one nearest to the base and so it starts computing its value before every other processor of the tile. Conversely, its top corner processor is the one furthest from the base and it terminates the computation of its value after every other processor. The simulation time for the full-tile is the same as the time between the virtual beat on which the bottom corner processor starts computing its value and the virtual beat on which the top corner processor terminates computing its in the virtual execution of the algorithm on the virtual array. Let d_b and d_t denote the distances of the bottom corner (virtual) processor from the base and of the top corner processor from the base. The bottom processor starts computing its value on virtual beat $3d_b/2$ while the top processor has terminated computing its on virtual beat $2d_t$. The time it takes to simulate the full-tile is thus given by $2d_t - 3d_b/2$. Since the full-tile is at tile level l, $d_b = s(q-l-1)+1$ and $d_t = s(q-l+1)-1$ and hence its simulation time is not more than $s(q-l+7)/2-3$ beats. There are l full-tiles on tile level l so the time to simulate all the tiles of the level is roughly $l(s(q-l+7)/2-3)$ beats. We can now determine the total number of beats required to simulate all the full-tiles of the array by evaluating the following summation:

$$\begin{aligned}
& \sum_{l=1}^{q-1} l(s(q-l+7)/2-3) \\
= \quad & (s(q+7)/2-3)\sum_{l=1}^{q-1} l - s/2 \sum_{l=1}^{q-1} l^2 \\
= \quad & ((s(q+7)/2-3)q^2/2 - s/2(q^3/3 - q^2/2 + q/6) \\
= \quad & sq^3/12 + 2sq^2 - 3q^2/2 - sq/12 \qquad (6.1)
\end{aligned}$$

Adding to this the $2qs$ beats required for the simulation of the base half-tiles we get the total number of beats required for the whole virtual array:

$$sq^3/12 + 2sq^2 - 3q^2/2 + 23sq/12 \qquad (6.2)$$

Only the most significant term of this expression is of interest to us. If we replace in this term q by n/s we get $n^3/12s^2$. Hence, the time complexity of the algorithm is $O(n^3)$. In term of complexity measure, we thus get no improvement over the sequential algorithm which also has a time complexity $O(n^3)$. What is interesting here is the $1/s^2$ factor in the constant hidden behind this complexity measure. The number of processors in the real array is s^2 so the fact that the running time of the algorithm is proportional to $1/s^2$ implies that in real term the speedup provided by the algorithm is optimal. In other words, building a bigger array will in most cases be worth it. This is not a new result. The optimal speedup property is inherent in the basic algorithm of K-GKT and this was pointed out by Guibas, King and Thompson in their paper. The result also applies to the recognition phase of my extension.

Chapter 7

Implementation issues

7.1 Introduction

Up until now I have spent my time in the clouds, in the very comfortable world of theory, a world where reality is as we define it to be, a world where cumbersome details can simply be swept under the carpet and forgotten. In this chapter, I make a brief visit into the world of practice and consider some issues related to the implementation of my algorithm. The chapter is divided into three parts. In the first part, I discuss the possible implementation of the CYK combination using PLAs. In the second, I describe three currently available machines (the Transputer, the Connection Machine, the DAP) and I discuss their suitability for the algorithm. Finally, in the last part, I report on an actual implementation (on the DAP) of my algorithm.

7.2 The CYK combination and parallelism

The main operation performed by the processor during the recognition phase is the CYK combination (see section 2.3.1). The operation has inherent parallelism which can be exploited through recourse to a broadly used construct in VLSI, the Programmable Logic Array (PLA) [Mead 80]. A PLA computes sums of products of boolean variables. It consists of two abutted rectangular areas called the **and**-*plane* and the or-*plane*. The input lines (i.e. the lines of the input variables) traverse the and-plane where they cross lines called the and-*lines*. Each input is inverted and both a line carrying the input's value and one carrying its complement cross the and-lines. Connections can be placed where these lines cross that will leave an and-line at high voltage only if an input is high or only if it is low, depending on whether the and-line is connected to the input's line or to its complement. The and-lines run across the whole PLA. In the or-plane they cross the output lines. Connections can also be placed at these crossings. A high and-line crossing an output line to which it is connected will put this output line low. An output line can remain high if and only if none of the and-lines to which it is connected is high. The signal on such an output line thus represents the inverse of a sum of products. If we want the sum of products, we invert the signal on the line. The placement of the connections in the PLA determines the boolean function it implements, hence the term "Programmable".

The CYK combination is a boolean function which takes as input two sets of non-terminals and produces one set. The presence of a non-terminal in a set is a boolean value. We can thus implement the CYK combination with a PLA. The input to such a PLA would be the two sets of signals representing the two input sets of non-terminals and the output would be the set of signals representing the CYK combination over these sets. Such an implementation would be extremely fast. The

time it would take to perform the operation would be of the order of magnitude of the time it would take to charge up a wire in the circuit. The idea of using PLAs has two major drawbacks. One is that it can require a prohibitively large amount of circuit space and the other is that it would make it difficult to change grammars. The straightforward design of the PLA (i.e. without optimisation) implies as many and-lines as there are right-hand sides in the grammar and a total number of input and output lines equal to five times the number of non-terminals in the grammar. Such a PLA would be very sparse. Since the grammars are in CNF, each and-line would be connected to only two non-terminals. There exist compaction techniques for PLAs such as folding [De Michelli 83] and partitioning [Cole 84]. But the application of such techniques rarely reduces the space below 30%. However, it could be the case that PLAs designed for the CYK combination are very good candidates for compaction. Whether this is true or not will depend on the specific grammar used. The second drawback is related to the fact that a PLA is a hardwired piece of circuitry. If the CYK combination for a particular grammar is implemented by a PLA, changing the grammar implies redesigning the PLA and thus redesigning and refabricating the whole chip containing the PLA. There is a way out of this problem. Instead of PLAs, we can resort to other programmable hardware devices such as EPROMs (Erasable Programmable Read Only Memories) or EPLDs (Erasable Programmable Logic Devices) and obtain nearly the same speed performance as with PLAs. But such constructs require a lot of space, at least as much space as unoptimised PLAs. So it is a case of "out of the frying pan, into the fire".

7.3 A look at 3 different machines

In this part of the chapter I take at look at 3 commercially available parallel machines. I first describe briefly each machine. I then point out some of their respective advantages in relation to my algorithm.

7.3.1 The Transputer array

The Transputer from Inmos Limited [Inmos 86] is a computer on a chip. It contains a powerful 32 bit processor, a 2 K byte RAM and four bidirectional serial asynchronous communication links (so called "Occam links") that can transmit data at a rate of up to 20 M bits/sec.. It can also access external memory. Its total address space (internal and external memory) can extend to upto 4 G bytes. The communication links are of the twisted wire type. To connect two Transputers all that is needed is two wires (channels), one for each direction of communication. The communication protocol does not allow uni-directional links. Via the links, information (i.e. messages) can be transferred from the address space of one Transputer to the address space of another. The four links and the processor can all operate concurrently.

A Transputer can act as a single process or as many. The concept of a process is embedded in the hardware. Processes running on different Transputers run concurrently while those running on a single Transputer share the time of the processor.

Transputers can be interconnected in very many ways. The fact that a Transputer has four communication links makes it very straightforward to build a rectangular array of Transputers.

7.3.2 The Connection Machine

The Connection Machine from Thinking Machine Inc. [Hillis 85] is a collection of 65,536 (2^{16}) 1 bit processors, each having its own local memory of 4096 bits. The processors all execute the same instruction stream. The machine is thus of the Single Instruction Multiple Data (SIMD) type. The stream of instructions is provided by a front end computer. Through a 1 bit flag internal to each processor, called the *context flag*, it is possible to deactivate one or many processors for the duration of one or many instructions. Only active processors execute instructions. (Except for the instruction that either sets or resets the context flag. Otherwise, the processors would imitate even more closely the cells of our brain, all slowly dying as time goes by...) The processors are interconnected by two different networks. One is a rectangular grid while the other is a 16-dimensional hypercube. Through the first, processors can communicate directly with each of their four nearest neighbours while through the second, any processor can communicate with any other processor in a maximum of 16 steps. The hardware allows the memory of each processor to be divided in equal portions and to allocate each portion to what is called a *virtual processor*. There may be up to 16 virtual processors per physical processor and so there may be in total up to 1,048,576 virtual processors. Notice that the comments above relative to the execution of instructions apply to virtual processors. Hence, each virtual processor has its own context flag. Only physical processors operate concurrently. Instruction execution by virtual processors must be multiplexed in time. Notice also that the comments above relative to inter-processor communication apply to virtual processors as well. Communication with the outside world can be performed in two ways, either via the front end processor or via an I/O bus. The front end can access the memory of a single processor or of a series of contiguous processors. Communication via the front end is relatively slow. The I/O bus, which can transfer data at a rate of up to 500 M bits/sec, is

used when information must be moved quickly to and from memory, for example, during disk swapping.

7.3.3 The DAP

There have been two generations of the DAP (Distributed Array of Processors). I first describe here the machine of the earlier generation, the International Computer Limited (I.C.L.) DAP [Gostick 79] [Reddaway 79] (first produced in 1976, first delivered in 1980). This is the one I have worked with. The I.C.L. DAP is a 64×64 rectangular array of 1 bit processors that each have a 4096 bit local memory, as in the Connection Machine. As in the Connection Machine, the processors all obey a single stream of instructions, each processor has a flag (which is called by the DAP people the *activity bit*) that indicates whether it will ignore or not the next instruction and each processor can communicate, 1 bit at a time, with each of its four nearest neighbours. The Distributed Array of Processors is called "Distributed" because the processors of the array are embedded in part of the core memory of an I.C.L. mainframe (Series 2900) host. The DAP has a central controller which fetches the instructions from the memory and broadcasts them to the 4096 processors of the array. The DAP controller is itself under the control of the host's CPU. To run a program on the DAP one first has the host load the program in the part of its memory which is also the DAP's memory. Once the program is loaded we then have the host pass control to the DAP's controller. At the end of the DAP program execution, control is transferred back to the host CPU. The DAP does not have its own I/O facility. It relies on the host for its I/O. This implies that whenever an I/O operation must be performed in a DAP program, the program must be interrupted and control must be passed to the host. The memory accessible to the DAP controller is the same as the memory of its processors.

Hence, a DAP program occupies a portion of each of its 4096 processors' 4096 bit local memories.

Active Memory Technology Ltd[1] (A.M.T.) took up the basic DAP architecture and started delivering second generation DAPs in 1985 [Parkinson 88]. These differ from the earlier models in the following way. First, the machine stands as an independent unit instead of being embedded in the memory of a mainframe. It is attached to a host via a 2 M bytes/sec interface. The host is used mainly for loading programs into the DAP and to initiate their execution. It does not provide the instruction stream as does the front-end of the Connection Machine. The local memory of each processor is 8 times bigger then in the first version DAPs (i.e. 32 K bits). Also, programs executed by the A.M.T. DAP reside in a memory unit separate from the processors' memory. Hence, the local memory of each processor really is "local". There exists a fast I/O facility through which data can be written or read, at the rate of 70 M bytes/sec, from a "plane" of the processor array memory called the D plane. The plane consists of 1 bit registers, one from each processor. Finally, as well as being connected to their 4 nearest neighbours, the processors are connected to buses that run along the rows and columns of the array. At the time of writting, A.M.T. offers a 32×32 DAP. A 64×64 one is expected to be available in the near future.

[1] A spin-off from I.C.L.

7.4 Pros and cons

In this section, I consider some aspects of the architecture of the machines described above that are, to a greater or lesser extent, relevant to my algorithm. I compare the machines in the aspects considered. The comparisons I make are mostly qualitative. I have not tried to measure or evaluate the performance of each machine. This could be the subject of some further work.

7.4.1 Computation/communication power balance

An important issue in the design of parallel computers composed of a large number of processors (so called *massively* parallel computers) is the balance between the performance of each processor and the performance of the interconnection network. If the processors are too powerful they will spend most of their time waiting for the terminations of data transfers and conversely if the network is too efficient it will be under utilised. A straightforward rectangular array of Transputers is an example of a machine where most of the hardware is devoted to processing power. The Connection Machine lies at the other extreme of the spectrum. Whether, for a particular machine, a balance is reached between its computation and communication capabilities will depend on the algorithm we want to run on the machine. Most systolic algorithms require relatively little computation power compared to the amount of data exchange they involve. This is also the case for my extension to K-GKT if we apply it to dynamic programming problems other then CFL parsing. For example, if we use the extension for the building of optimal binary search trees, each beat of the first phase of the algorithm will involve for each processor only a few additions, a couple of comparisons and perhaps the saving of counter values. If we use it for CFL parsing, the main operation performed during each beat of the

first phase (the recognition phase) is the CYK combination. This operation can be quite elaborate. How elaborate will depend on the grammar used. Two attributes of a given grammar will affect the computation/communication balance. One is the size of the set of non-terminals of the grammar and the other is the size of its set of rules. Since non-terminal sets are exchanged by processors, a grammar with a larger number of non-terminals will incur heavier inter-processor communications. Since sets of non-terminals must also be stored in the processors a larger number of non-terminals will also imply bigger processor memories. Since the complexity of the CYK combination is directly proportional to the number of rules in the grammar, the number of rules will influence the processing requirement of the processors in the array. We may thus say that the balance reached between the computation and communication power of the machine using a given grammar will depend to some degree on the balance between the sizes of the grammar's non-terminal and rule sets.

Notice that if, for an algorithm, the processors of a machine are too powerful in comparison with the performance of its inter-processor communication network, we can, in order to make a more balanced usage of the machine's computation and communication capabilities, have each of its processors emulate a square sub-array of virtual processors. In doing so, we will shift a part of the communication load from the inter-processor links to the processors themselves since the latter will perform (emulate) the data transfers that are to occur between its virtual processors. To a limited extent, this strategy can also be used the other way around. Suppose that for a given algorithm, the storage capacity of the processors of a machine is insufficient. We can in that case cluster processors together and have each cluster emulate one virtual processor. Some of the data references performed by such a virtual processor in its virtual storage would imply communication between processors of the cluster.

7.4.2 Code replication

My parsing extension to K-GKT is nearly an SIMD algorithm. An array of Transputers is a Multiple Instruction Multiple Data (MIMD) type machine. We can still have the extension run on such an array. Doing so implies that the program (and the grammar description) has to be loaded in the local memory of each Transputer. It almost seems a waste of space to replicate the code in such a way. In contrast, if we run the algorithm on the Connection Machine or the DAP, we will have just one copy of the code (and of the grammar). On the other hand, observe that with these true SIMD machines, we have to broadcast to all the processors each instruction that needs to be executed. Since one single instruction might be executed many times, it almost seems a waste of time to broadcast it to all the processors each time it is executed rather than to load it into their memory for once and for all. So, the choice between replicating the code in each processor's memory or broadcasting it to all the processors can be seen as a space/time tradeoff.

My extension to K-GKT is not a pure SIMD algorithm. For example, in the parse output phase, tree-node processors and link-node processors behave slightly differently. In relation to this aspect of the algorithm, the MIMD approach has an advantage over the SIMD approach. On an MIMD machine such as a Transputer array, the execution of the code of a tree-node processor can overlap completely with the execution of the code of a link-node processor while on an SIMD machine such as the Connection Machine or the DAP an overlap can occur only on segments of code that are common to both processors. How significant is the advantage of an MIMD machine over an SIMD machine in relation to my algorithm? Because the algorithm is systolic, all the processors have to be synchronised on equal length beats. The beat period must be at least as long as the time it takes for the slowest processors to do their work. Hence, since during the output phase there are only 2 types of (non-idle) processor (tree-node and link-node), the speedup that can

be obtained during the phase by overlapping the code execution of each type of processor cannot be greater than 2. Actually, the work performed by a link-node processor during a beat, sending a value upward and obtaining one from downward, is also a major portion of the work performed by a tree-node and so the speedup should be nearer to 1 than to 2. In the other two phases, recognition and marking, there is no type distinction between processors. So, in the context, the MIMD approach does not bring important benefits over the SIMD approach.

7.4.3 The number of processors

As was already said, a Transputer is a very sophisticated processor compared with a Connection Machine processor (or a DAP processor). Consequently, the cost of building an array of Transputers containing as many processors as in the Connection Machine (or the DAP) will be much greater than the cost of building a Connection Machine (or a DAP). (I am not taking into account here the cost of the hypercube network of the Connection Machine. Observe that this network is of no use to us.) Since the number of processors required by the algorithm is quite large ($O(n^2)$), this could be a predominant factor in the choice of a machine. On the other hand, recall that it is possible to have one Transputer emulate many virtual processors so it is possible to have a moderate size array of Transputers emulate a large size array of virtual processors. (Notice that we can also have Connection Machine or DAP processors emulate virtual processors.)

7.4.4 I/O and problem decomposition

Of the 3 machines I have described, the Transputer array is the one most suited for the implementation of the scheme described in chapter 6 for problem decomposition. In this scheme, processors at the boundary of the array need to write

to (or read from) auxiliary memory (disk) at the same rate as processors within the array exchange data. The hardware links on the Transputers that are used for inter-Transputer communication can also be used for communication between Transputers and peripherals. Communication of the latter type can take place concurrently with communication of the former. On the Connection Machine or the DAP it is possible for the front end to have access to the local memory of selected processors. However, the exchange of data between the front end and the processor array is slow and it cannot overlap with processing. Both the Connection Machine and the A.M.T. DAP have a fast interface allowing the loading of data in or the reading of data out of the whole machine's memory very quickly. Unfortunately, through these interfaces, it is not possible to access the memory of only the border processors of the arrays.

7.4.5 A triangular array

The algorithm uses a triangular array of processors. The DAP and the Connection Machine are rectangular arrays. Hence, when using them, we can exploit at most only (slightly over) half of their processors. If we were to build a Transputer array especially for our algorithm, it would be easy to tailor the array to our needs, i.e. to make it triangular. "But," one could say, "so would it be the case if we were to build a DAP-like machine especially for our algorithm." The flexibility of the Transputer however would make the task particularly easy.

7.5 An actual implementation

I have written an implementation of my extension to K-GKT for the I.C.L. DAP (first generation DAP). The whole implementation consists of two separate programs. A copy of these is reproduced in appendix A. My primary goal in writing the implementation was not so much to evaluate the performance of my algorithm but rather to test it and try to find out if it did not contain any flaw that I had overlooked. Writing a program that supposedly implements an algorithm and verifying that the program produces the expected results does not constitute a valid proof of the algorithm's correctness. However, writing the program and seeing that it produces bad results can certainly help in pinpointing errors in the algorithm.

When I wrote the implementation (summer 1986) the University of Edinburgh had an I.C.L. DAP of 64×64 processors[2]. A few months later the University also acquired an array of Transputers produced by Meiko[3] called the Computing Surface. It would have certainly been interesting to implement my extension on the Computing Surface as well but one implementation was sufficient to reach the goal mentioned above. Also, the fact that at the time the Computing Surface consisted of only 40 processors made it less appealing (to some macho parallel computer programmer) than the 4096 processor DAP.

---

[2]Which had been donated to the University in 1983 in celebration of its 400th year of existence by Queen Mary College, University of London.

[3]Meiko Limited, Whitefriars, Lewins Road, Bristol, UK.

7.5.1 An ad hoc parser for $L(G_1)$

As the aim of the programming experiment was simply to test my algorithm, I did not consider it worthwhile to write a general parser and wrote instead a very ad hoc parser for G_1 (see page 22). The fact that G_1 is efficient made the implementation simple in that it implied that each processor needed to store at most one pointer pair. The fact that G_1 is *rhs*-disjoint but not *nt*-disjoint made the implementation interesting in that it implied that in the marking phase, non-terminal sets rather than just simple tokens had to be exchanged.

7.5.2 DAP Fortran

The programs have been written in a special Fortran extension for the DAP called DAP Fortran. I shall not bore the reader here with the details of the code. The programs are rather heavily commented and self-explanatory. I simply explain here a few important features of DAP Fortran.

Host and DAP components

As I have mentioned in section 7.3.3, the I.C.L. DAP is under the control of a host machine (I.C.L. Series 2900). To execute a program on the DAP, one writes two programs, one for the host and one for the DAP itself. The DAP component is not a program as such but rather a set of Fortran subroutines. Some of these are "Entry subroutines" which are callable from the main host component program. The two components communicate with each other via Fortran "Common blocks". Only the DAP component is written in DAP Fortran. The host component is written in standard Fortran. In the implementation, the host component, apart from calling the relevant subroutines of the DAP component, does mundane things

such as prompting the user for an input string and displaying the result of the parse in some fancy format.

DAP Fortran matrices

To express in DAP Fortran operations that are to be performed by all the DAP processors (or a subset thereof), the user writes statements involving 64×64 matrices. For example, to declare a 5 bit vector (called `FLAGS`) in each of the 4096 processors of the DAP, one would write the statement:

```
LOGICAL FLAGS (,,5)
```

This statement contains two *implicit indices*, `(,,5)`, which indicate to DAP Fortran that this is a $64 \times 64 \times 5$ array of bits or, equivalently, a 64×64 matrix of 5 bit values. (One could also declare matrices of integers, of reals, or of characters in the same fashion.) DAP matrices can appear in Fortran expressions and on the left-hand side of assignment statements. Operations involving DAP matrices are performed in parallel on each of the matrices' elements. For example, the following statement has each of the 4096 DAP processors compute the OR of the first and second bit of their local `FLAGS` variable and assign that value to the third bit of this variable.

```
FLAGS(,,3) = FLAGS(,,1).OR.FLAGS(,,2)
```

The 4096 computations and the 4096 assignments are all carried out in parallel.

Masked assignments

The implicit indices of a matrix variable reference that appears on the left-hand side of an expression can be replaced by a boolean expression involving DAP matrix

variables (only). Such an expression serves as a mask. Its effect is to have all the processors compute locally a boolean expression and have only those processors that have computed a true value carry out the assignments. For example, the following statement has those processors whose `FLAGS(1)` or whose `FLAGS(2)` bit is set assign to their `FLAGS(3)` bit the value of their `FLAGS(4)` bit.

```
FLAGS(FLAGS(1).OR.FLAGS(2),3) = FLAGS(4)
```

All the other processors leave their `FLAGS(3)` bit unchanged.

Inter-processor communications

Communication between DAP processors is expressed in DAP Fortran by expressions involving *shifted* DAP matrices. An implicit index in a reference to a DAP matrix can be replaced by either a "+" or a "-" to indicate that processors are to use a matrix element coming from a neighbouring processor. For example, a "+" as the implicit row index of a DAP matrix and an empty implicit column index indicate that the processors are to obtain the matrix element value from their bottom neighbours. As another (fancy) example, the following statement has those processors whose top right neighbour's `FLAGS(1)` bit is set assign to their own `FLAGS(2)` bit the value of their left neighbour's `FLAGS(3)` bit.

```
FLAGS(FLAGS(-,+,1),2) = FLAGS(,-,3)
```

One of two things can happen to processors on the edge of the array. They can either be connected in a wrap around fashion to processors at the opposite edge (`CYCLE` mode) or they can be connected to *virtual* neighbours that are to feed them only with null values (`PLANE` mode). The user specifies which modes he requires through the `GEOMETRY` statement. Separate modes can be specified for each of the two directions, vertical and horizontal. The following statement for example

specifies that left or right boundary processors are connected to processors on the opposite edge while processors on the top or bottom boundary are connected to virtual processors.

```
GEOMETRY(CYCLE,PLANE)
```

For my implementation, I used:

```
GEOMETRY(PLANE,PLANE)
```

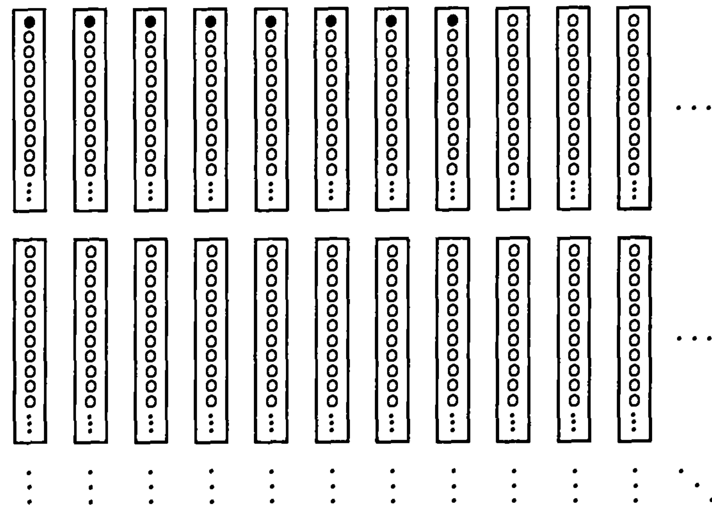
Storage modes

The storage mode of matrices of the host Fortran differs from that of the DAP Fortran. For example, in host Fortran storage mode an integer of a 64×64 8 bit integer matrix occupies 1 bit in the local memory of each of 8 contiguous DAP processors. In DAP Fortran storage mode, *the same* integer should occupy 8 contiguous bits in the local memory of 1 DAP processor (see figure 7–1). In either mode, the whole matrix occupies exactly the same bloc of 32768 bits. If the host component of the program assigns values to elements of a matrix, this matrix will have to be converted before it can be processed by the DAP and vice versa. DAP Fortran has predefined subroutines to perform the required conversions. These all work *in situ.* Matrices of single bits (`LOGICAL`) are not affected by the storage mode difference. The matrix `FLAGS` of our examples above is affected.

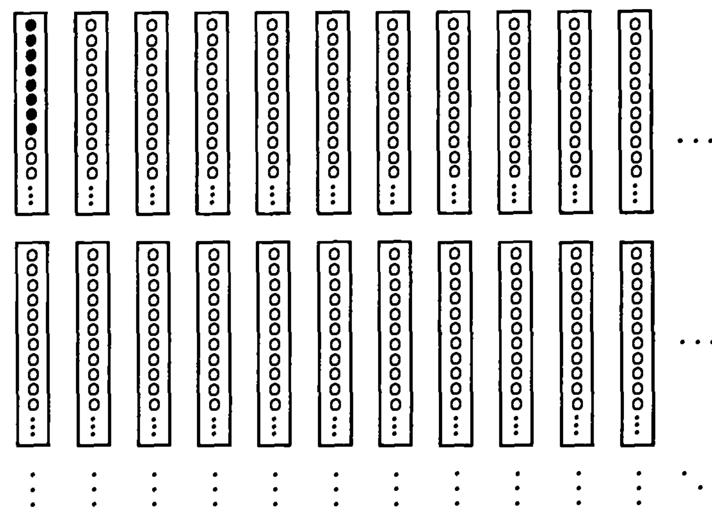
7.5.3 The implementation

AETKHOST and AETKDAP

Appendix A contains the two components of the implementation. The first one, called AETKHOST (AETK stands for An Extension to K-GKT) is the host component and the second, called AETKDAP, is the DAP component. Perhaps only



a) Host Fortran storage mode



b) DAP Fortran storage mode

Figure 7–1: The storage of an 8 bit integer in Host Fortran and DAP Fortran.

two aspects of the programs are worth mentioning here. First, the implementation was developed in 4 phases. In a first phase I implemented the basic K-GKT algorithm (more specifically, the GKT version). In a second phase I added counters and pointers. In a third phase I implemented the marking and finally in the last phase, I added the parse output phase of my extension. The comments in the program bear a trace of this stepwise development[4]. Second, I chose as the top of the (tilted) array the top left corner of the DAP array. Hence, while throughout the text the processor indexing convention I use is consistent with the idea of an upper right triangular array, in the programs I switch to an upper left triangular array[5]. Notice that one could also use a lower left or a lower right triangular array. The orientation of the array is of no importance, as long as consistency is ensured.

Performance evaluation

The host component program contains a bit of code intended to measure the execution time of the DAP. However, I was unable to obtain significant measures due to the simple fact that the predefined function `DAPTIME` returned an integer value expressing the time in seconds. The DAP never took more than 1 second to execute the algorithm, whatever the length of the input was (the latter could not go beyond 64 tokens). Due to the low accuracy of the `DAPTIME` function, I could not know how much time below 1 second it took. I could have gone around this problem however by having the DAP execute repeatedly several hundred times or

---

[4]In the comments, the phases are numbered from 0 to 3. Phase 0 is never mentioned as such.

[5]Some reader will want to know **why**. It is quite simple. The "top" of the array being the top left corner, I always expressed data movement "up" the array, whether in the row direction or column direction, with the "-" sign as a DAP (*shifted*) matrix implicit index. Had I stuck to the upper right convention, I would have had to use one sign for one direction and another sign for the other.

several thousand times the algorithm (on the same input) and measure the time it took. This is a rather simple idea. Unfortunately it did not come to my mind in time (before the DAP was dismantled). As a final comment on performance let me point out that probably a lot of overhead can be attributed to the conversions (from Fortran storage mode to DAP Fortran storage mode) that had to be performed on the DAP matrices.

A straightforward affair

As I have stated at the beginning of this section, the main purpose of writing the implementation was to test the algorithm and to expose some of the faults it may have had. The writing of the programs was most straightforward. The main difficulty encountered, which had nothing to do with the algorithm, consisted in getting the storage mode conversion function calls right. A minor difficulty was getting acquainted with DAP Fortran. Another difficulty was getting reacquainted with Fortran after 8 years separation. Writing the DAP programs was good fun!

Chapter 8

CIP and parse orders

8.1 Introduction

I have mentioned in chapter 1 the work of Chang, Ibarra and Palis whose algorithm (CIP), like mine, maps on an array of processors the recognition phase and the parse extraction phase of the CYK algorithm. In the description of their algorithm, Chang et al. show how to obtain a rightmost (or reverse rightmost) parse of the analysed string. In this chapter, I show how we can adapt the CIP algorithm, as I have adapted mine, to produce parses in orders other than rightmost. Recall that the CIP algorithm runs on an array of processors composed of two sub-arrays, the recognition sub-array and the parse extraction sub-array. When run on a valid input, the algorithm produces, in the parse extraction sub-array, a distributed representation of the parse tree of the input. The algorithm then proceeds by flattening this representation so as to obtain, in the bottom processors, a post-order enumeration of the tree's nodes. I introduce two variants of this flattening phase. One variant flattens the tree into the preorder enumeration while the other flattens it into an inorder enumeration.

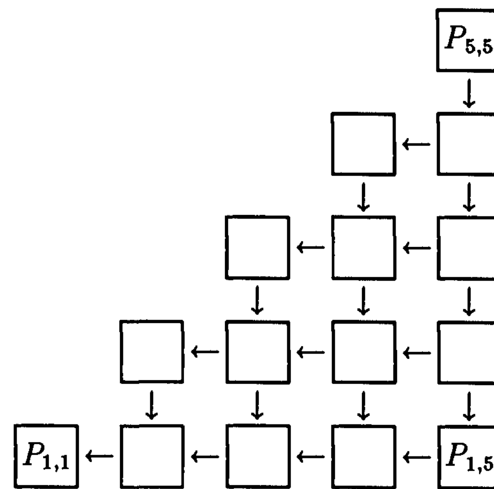


Figure 8–1: The CIP parse extraction sub-array.

I first define formally what I refer to as a *valid tree representation* in a parse extraction array. I then proceed by describing the variants and by proving them correct.

8.1.1 Valid tree representation

For the purpose of this discussion, I number the rows of the parse extraction array from bottom to top and the columns from left to right. Hence, for an $n \times n$ array, $P_{1,1}$ is the lower leftmost processor, $P_{1,n}$ is the lower rightmost processor and $P_{n,n}$ is the uppermost one (see figure 8–1). I call row 1 of the array its *base* and I define the size of the array to be its number of rows (or columns). The processors of a parse extraction array may either hold a *null* or *non-null* values. As we will see, a processor holding a non-null value corresponds to some node of a binary tree. For this reason, I refer to such a processor as a *tree node processor.* The value held in the processor corresponds to the label of the node.

Definition 8.1.1 *A parse extraction array holds a* valid tree representation *if it satisfies the following four conditions:*

1. *The topmost processor of the array is a tree node processor (it holds a non-null value). It corresponds to the root of the tree represented.*

2. *The base processors are all tree node processors. They correspond to the leaves of the tree.*

3. *If the nearest tree-node processor R below some tree node processor F on row D $(D > 0)$ is on row d, then the nearest tree node processor L in the south-west direction from F is the one on row $D - d$. F corresponds to a father node and L and R to its left and right sons.*

4. *The root, processor $P_{n,n}$, is the ancestor of every tree-node processor in the array.*

The following theorem points out the one-to-one correspondence existing between valid tree representations in parse extraction arrays and full binary trees. In this theorem and in its proof, I abstract the specific values held in processors to being either null or non-null.

Theorem 8.1.1 *A valid tree representation held in an array of size* n corresponds *to some full binary tree of $q = 2n - 1$ nodes and vice versa.*

Proof I first establish the correspondence from valid tree representations to full binary trees and then I establish the correspondence in the opposite direction.

1— On induction on n, the size of the array. The basis is true for $n = 1$. The array then consists of one processor holding a non-null value and it corresponds to a one node full binary tree. Assuming the hypothesis holds for any array of size

less than n for some $n > 1$, I prove that it also holds for any array of size n. Let us consider some array of size n holding a valid tree representation. By definition 8.1.1 its topmost processor is a tree node processor. Since its base processors are also tree node processors (by definition 8.1.1), the topmost processor must have a left son L (nearest south-west tree node processor) and a right son R (nearest tree node processor below). Suppose the left son is on row n_l. Then, by definition 8.1.1, the right son must be on row $n - n_l$. Consider the two lower right triangular sub-arrays whose topmost processors are L and R. These sub-arrays each satisfies all four conditions of definition 8.1.1. Since they are of size less than n, by the induction hypothesis, they correspond to full binary trees of respectively $2n_l - 1$ and $2(n - n_l) - 1$ nodes. Since the roots of these trees are the left and right sons of the topmost processor of the whole array, the whole array thus corresponds to a full binary tree of $2n - 1$ nodes.

2— By induction on q, the number of nodes in the full binary tree. For $q = 1$, the tree consists of only 1 node and it corresponds to an array of 1 tree node processor. Such an array satisfies all four conditions of definition 8.1.1 and hence the hypothesis is true for $q = 1$. Suppose the hypothesis is true for any full binary tree of less than q nodes for some $q > 1$. Let us prove that it also holds for any tree of q nodes. Such a tree has two sub-trees of less than q nodes which, themselves, are full binary trees. Suppose the left sub-tree has q_l nodes. Consequently, the right sub-tree must have $q-q_l-1$ nodes. Let q_r denote $q-q_l-1$. The tree and its sub-trees being full binary trees, q, q_l and q_r must all be odd. By the induction hypothesis, to the left and right sub-trees correspond valid tree representations fitting in arrays of size $(q_l + 1)/2$ and $(q_r + 1)/2$ respectively. We can construct an array of size $(q + 1)/2$ by abutting these two arrays side by side and by adding processors in between the arrays and above the array corresponding to the right sub-tree. If all the added processors except the topmost one hold null values, we obtain an array corresponding to the whole tree. The topmost processor corresponds to the root of

the tree and the processors corresponding to the left and right son of this topmost processor, by definition 8.1.1, are just those processors corresponding to the roots of the left and right sub-trees, by the induction hypothesis. The resulting array satisfies all four conditions of definition 8.1.1. ■

8.2 Preorder enumeration

I now define an algorithm to flatten the tree represented in an array onto the bottom processors, into a preorder enumeration. The algorithm is similar to the one of CIP. The latter has the processor values (rules) migrate directly downward towards the base processors and it shifts values in overcrowded base processors leftward. The following algorithm instead has the values migrate diagonally (downward and leftward) and it shifts extra values in base processors rightward.

Algorithm 8.2.1 *Base processors have two registers denoted MR (most recent) and LR (least recent). The base processors' initial values are stored in their MR registers and their LR registers initially contain null values. On odd beats, every processor not on the base sends the value it holds to its bottom neighbour. On even beats, every processor including those on the base sends the value it received on the previous (odd) beat from above to its left neighbour. Null values are fed on the top-left and right boundaries of the array. Whenever a base processor receives either from the left or from the right a non-null value, it pushes this value into its MR register. The value that was in the MR register is pushed in the LR register and if the LR register already held a non-null value, this value is sent to the right on the next beat.*

On a $n \times n$ array, the algorithm terminates after $3n - 2$ beats. The detection of termination can easily be implemented using a control signal. At the end of the

algorithm's execution, the leftmost enumeration sought is stored from left to right in the MR and LR registers (in this order) of the base processors. This enumeration (if it is a parse) can be output sequentially via the leftmost base processor to give a leftmost order parse or via the rightmost base processor to give an inverse leftmost order parse. Figure 8–2 shows the state of the parse extraction array on various beats during the execution of the algorithm on the example of section 2.3.6. I now formally show that this algorithm performs as expected. In the following theorem I let LR_j and MR_j denote the LR and MR registers of processor $P_{1,j}$.

Theorem 8.2.1 *For any n, $n \geq 1$, any j, $0 < j < n$, after beat $2n + j - 3$ of the execution of algorithm 8.2.1 on a $n \times n$ array holding a valid tree representation, MR_j and LR_j hold respectively the $(2j - 1)^{\text{th}}$ and $(2j)^{\text{th}}$ values of the preorder enumeration of the tree node and after beat $3n - 3$, MR_n holds the $(2n - 1)^{\text{th}}$ (last) value of the enumeration. At all times, LR_n holds a null value, processor $P_{1,1}$ sends only null values leftward and processor $P_{1,n}$ sends only null values rightward.*

Proof By induction on n. The basis is true for $n = 1$. In that case the array consists of one processor which holds the representation of a one node tree. This processor simply keeps its own value in MR_1 and a null value in LR_1 from beat 0 onwards. Let us prove that the theorem is true for an $n \times n$ array for any $n > 1$ if it is true for any array of size less than $n \times n$. Let us consider an $n \times n$ array holding a valid tree representation. We can decompose the represented tree into its root, a left sub-tree and a right sub-tree. Suppose the left sub-tree has $2m - 1$ nodes and that consequently the right sub-tree has $2(n - m) - 1$ nodes. By definition 8.1.1 it follows that the left sub-tree is held in the leftmost $m \times m$ triangular portion of the array and that the right sub-tree is held in the bottom rightmost $(n - m) \times (n - m)$ portion. Let us call these portions respectively the *left sub-array* and the *right-sub-array* (see figure 8–3). By definition 8.1.1, it also follows that the processors in the rest of the array with the exception of $P_{n,n}$ all hold null values. Because

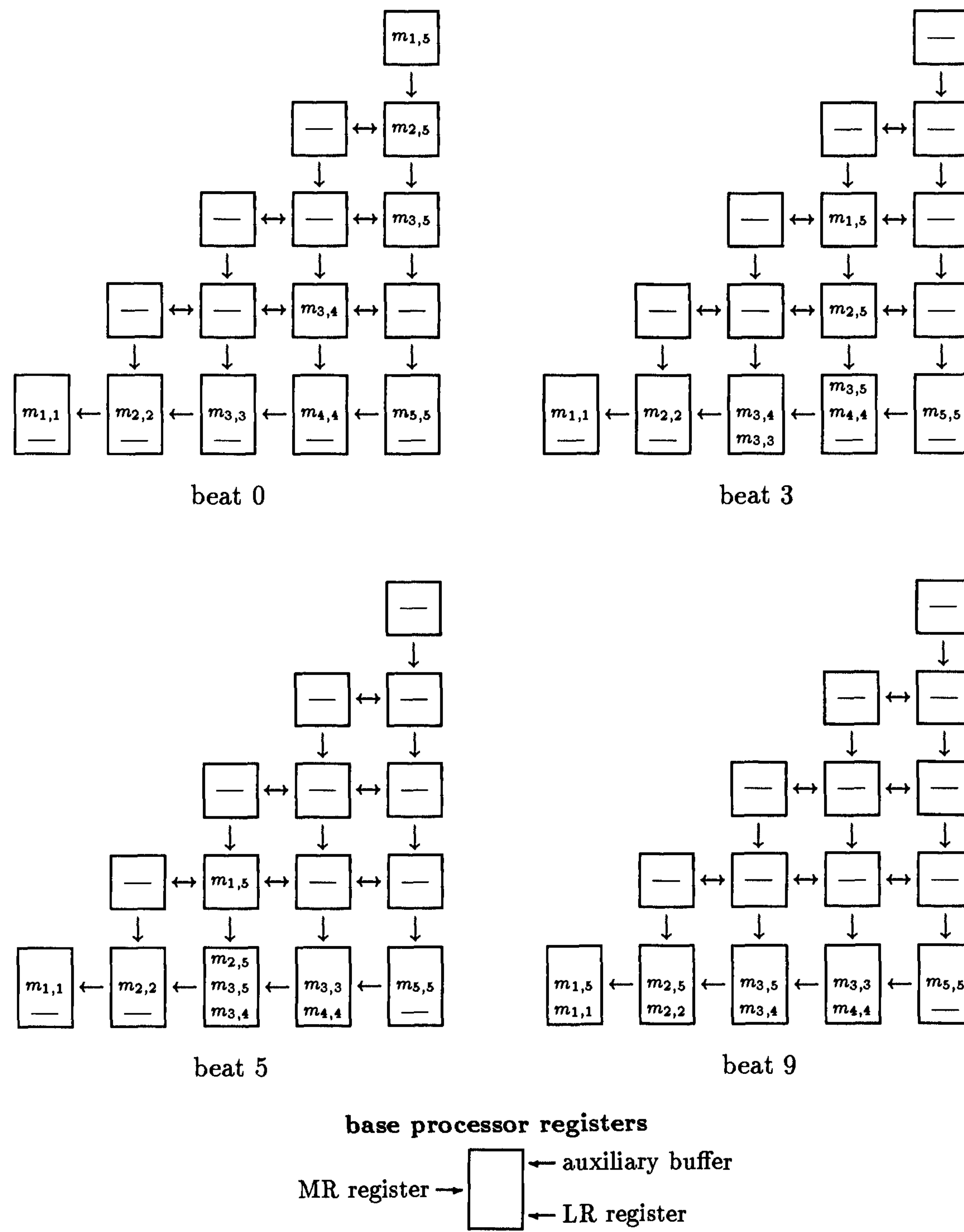


Figure 8–2: CIP flattening phase adapted for preorder enumeration.

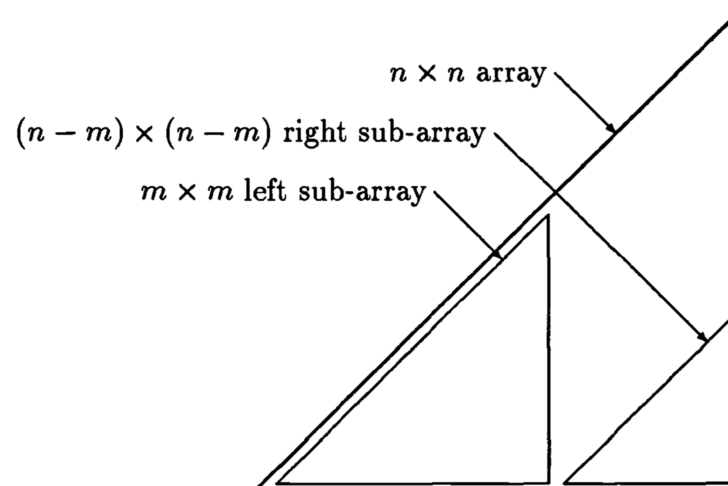


Figure 8–3: The sub-arrays relative to a tree in the array.

of this last fact and because the algorithm feeds null values on the top-left and right boundaries, we conclude the following. As far as the MR and LR registers are concerned, running the algorithm on the whole array has exactly the same effect as if we ran it simultaneously and independently on the left sub-array and the right sub-array (as if these were arrays on their own) except that in the former case, on beat $2n-2$ processor $P_{1,1}$ receives from the right a non-null value (while in the latter case it receives a null value). This is the value of the root node, i.e. the value of $P_{n,n}$. The effect of the arrival of this value will not be felt in $P_{1,1}$ before beat $2n-2$ and, in general, it will not be felt in any processor $P_{1,j}$, $1 \leq j \leq n$, before beat $2n+j-3$. We can thus apply the induction hypothesis in relation to either the left sub-array or the right sub-array and in relation to some processor $P_{1,j}$ up until beat $2n+j-3$. If the left sub-tree consists of only one node ($m=1$), then, by the induction hypothesis (applied in relation to the left sub-array), on beat $2n-2$, MR_1 contains the first and only value of the preorder enumeration of the left sub-tree (which is also the second value of the preorder enumeration of the whole tree) and LR_1 contains a null value. On that beat, $P_{1,1}$ receives the root

node value. This value goes in MR_1 and the old value of MR_1 goes in LR_1. Thus at the end of beat $2n-2$, MR_1 and LR_1 contain the first two values of the preorder enumeration of the whole tree. Afterwards, $P_{1,1}$ receives only null values from $P_{1,2}$ and thus MR_1 and LR_1 are never modified and $P_{1,1}$ sends only null values leftward and rightward (to $P_{1,2}$). Hence, the arrival of the root node values in $P_{1,1}$ has no effect on the MR and LR registers of the other base processors and so even after beat $2n+j-3$ the induction hypothesis applies in relation to the right sub-array and any processor $P_{1,j}$, $1 < j \leq n$. Thus for any j, $0 < j < n-1$, after beat $2(n-1)+j-3$, MR_{j+1} and LR_{j+1} contain the $(2j-1)^{\mathrm{th}}$ and $(2j)^{\mathrm{th}}$ values of the preorder enumeration of the right sub-tree (LR_n contains a null value). These correspond to the $(2j+1)^{\mathrm{th}}$ and $(2j+2)^{\mathrm{th}}$ values of the preorder enumeration of the whole tree and this completes the proof for the case where the left sub-tree consists of only one node. Let us now consider the more general case where the left sub-tree consists of $2m-1$ nodes for some m, $1 < m < n$. By the induction hypothesis applied to the left sub-array, when, on beat $2n-2$, $P_{1,1}$ receives the root node value from $P_{1,2}$, it holds in its MR and LR registers the first two values of the preorder enumeration of the left sub-tree. On the next beat the value that was in LR_1, which is the third value of the preorder enumeration of the whole tree, is sent to $P_{1,2}$ while the root node value and the value that was in MR_1, which are the two first values of the preorder enumeration of the whole tree, go in MR_1 and LR_1. From then on, $P_{1,1}$ receives only null values from $P_{1,2}$ since $P_{1,2}$ receives from beat $2n-1$ only null values from above. Hence the values in MR_1 and LR_1 remain as they are afterwards. By the same argument we can show that a process corresponding to the one just described takes place (propagates) in $P_{1,2}$ on beats $2n-1$ and $2n$ and in general, in $P_{1,j}$ on beats $2n+j-3$ and $2n+j-2$ for any j, $0 < j < m$. That is, on beat $2n+j-3$ processor $P_{1,j}$ receives the $(2j-1)^{\mathrm{th}}$ value of the preorder enumeration of the whole tree. At that time, MR_j and LR_j hold the $(2j)^{\mathrm{th}}$ and $(2j+1)^{\mathrm{th}}$ values of the enumeration. On the next beat, the $(2j+1)^{\mathrm{th}}$

value is sent to $P_{1,j+1}$ while the $(2j-1)^{\text{th}}$ and $(2j)^{\text{th}}$ values get stored in MR_j and LR_j. The contents of MR_j and LR_j remain unchanged afterwards. By the same process, on beat $2n+m-3$, $P_{1,m}$ receives the $(2m-1)^{\text{th}}$ value of the preorder enumeration of the whole tree. By the induction hypothesis MR_m then holds the $(2m)^{\text{th}}$ value of the enumeration and LR_m holds a null value. It follows that the $(2m-1)^{\text{th}}$ and $(2m)^{\text{th}}$ values get stored in MR_m and LR_m on beat $2n+m-3$. It also follows that $P_{1,m}$ will have sent and will send only null values to $P_{1,m+1}$. We thus conclude that as in the case where $m=1$, the induction hypothesis applies to the whole of the right sub-array. This completes the proof for the general case. ■

Corolary 8.2.1.1 *When run on an array holding a valid tree representation, algorithm 8.2.1 flattens the tree into its preorder enumeration onto the bottom boundary of the array.*

Proof Follows from theorem 8.2.1. ■

8.3 Inorder enumeration

The algorithm I am about to introduce resembles the original flattening phase of CIP even more than algorithm 8.2.1. As in CIP, the algorithm has the values migrate directly downward. If differs from CIP in its dealing with the arrival of a third non-null value in a base processor. Instead of getting rid of the least recently received value, it sends away the newcomer.

Algorithm 8.3.1 *Base processors have two registers denoted FI (first in) and SI (second in). The initial values of the base processors are stored in their FI registers and the SI registers start with null values. On every beat every processor except the base processors sends the value it holds to its bottom neighbour. Null values*

are fed to the processors on the top-left boundary. When a base processor receives a non-null value for the first time, either from above or from the right, it stores it in its SI register. Any non-null value that it receives thereafter, it sends to its left neighbour on the following beat.

When run on a $n \times n$ array, the algorithm terminates after (at most) $2n - 2$ beats. This is the time the root value would take to travel from the top of the array to the bottom-left corner. Figure 8–4 depicts the state of the parse extraction array at various beats during the algorithm execution on our example. At the end, the inorder enumeration of the tree's nodes is found in the SI and FI (in this order) registers of the base processors. The SI register of the leftmost base processor never receives a non-null value. A proof of the correctness of the algorithm follows. Not surprisingly, this proof is of the same shape as the proof of theorem 8.2.1. Feel totally free to skip it. In the following I have FI_j and SI_j denote the FI and SI registers of processor $P_{1,j}$.

Theorem 8.3.1 *For any n, $n \geq 0$, and any j, $0 \leq j < n-1$, after beat $n+j-1$ of the execution of algorithm 8.3.1 on a $n \times n$ parse extraction array holding a valid tree representation, SI_{n-j} and FI_{n-j} hold respectively the $(2(n-j)-2)^{\text{th}}$ and $(2(n-j)-1)^{\text{th}}$ values of the inorder enumeration of the tree and after beat $2n-2$, FI_1 holds the first value of the inorder enumeration of the tree. At all time SI_1 holds a null value and processor $P_{1,1}$ sends only null values leftward.*

Proof By induction on n. The basis is true for $n = 1$. In that case, the array consists of one processor which holds the representation of a one node tree. The processor simply keeps its own value in FI_1 and a null value in SI_1 and sends only null values from beat 0 onwards. Let us assume that the theorem is true for any array of size less than $n \times n$ and let us consider a $n \times n$ array holding a valid tree representation. As in the proof of theorem 8.2.1, I decompose the tree into its root,

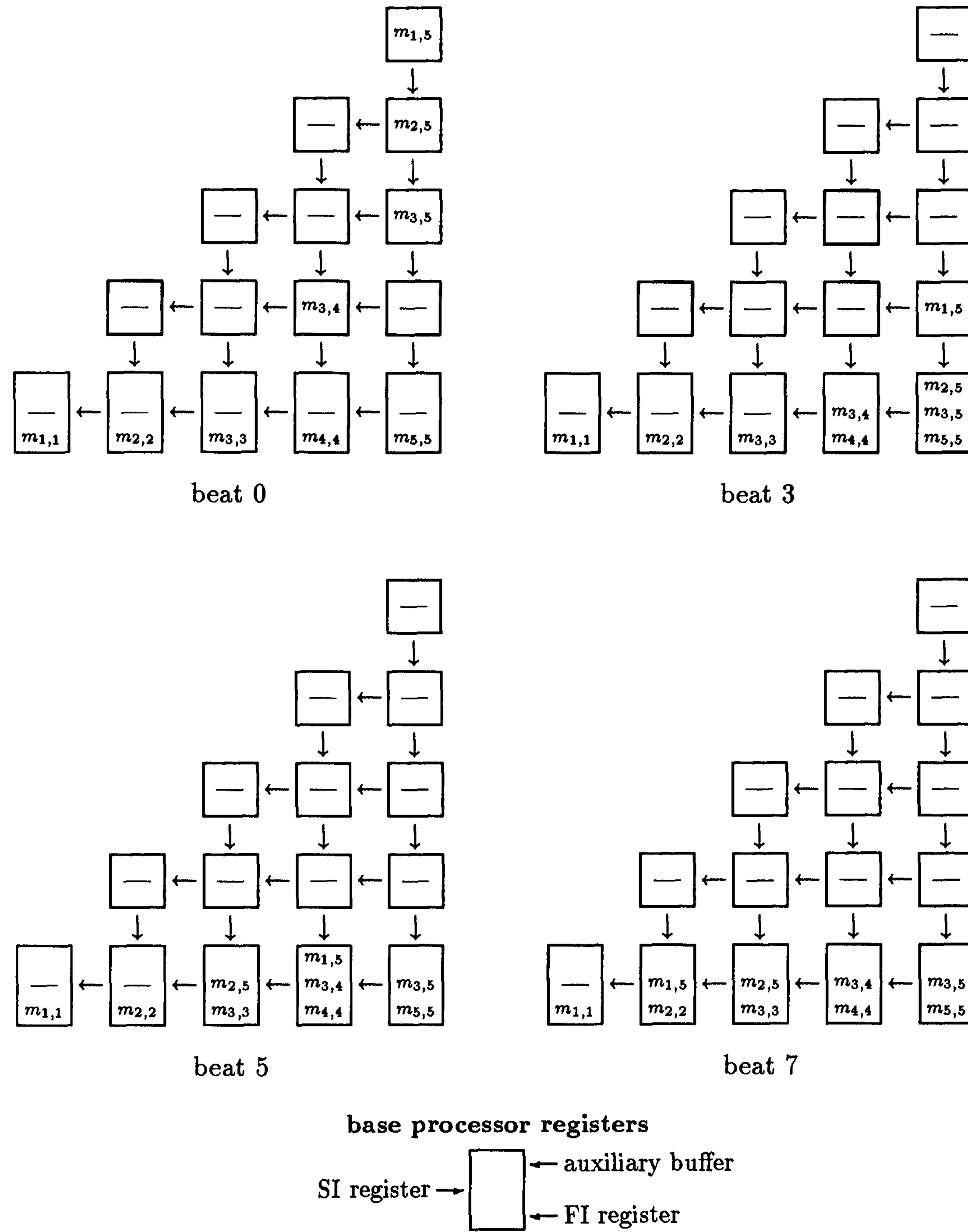


Figure 8–4: CIP flattening phase adapted for inorder enumeration.

a left sub-tree of say $2(n-m)-1$ nodes, $1 \leq m < n$, and a right sub-tree of $2m-1$ nodes. I also identify a left sub-array and a right sub-array and note that as far as the FI and SI registers are concerned, running the algorithm on the whole array has exactly the same effect as running it simultaneously and independently on the left and right sub-arrays (as if they were arrays on their own) except that in the former case, on beat $n-1$, processor $P_{1,n}$ receives the root node value while in the latter case it receives a null value. The effect of this cannot be felt in processor $P_{1,n}$ before beat $n-1$ and in general it cannot be felt in any processor $P_{1,n-j}$, $0 \leq j < n$, before beat $n+j-1$. We can thus apply the induction hypothesis in relation to either the left or the right sub-tree and in relation to some processor $P_{1,n-j}$ up until beat $n+j-1$. If the right sub-tree consists of only one node ($m = 1$) then, by the induction hypothesis (applied to the right sub-tree), on beat $n-1$, FI_n contains the first and last and only value of the inorder enumeration of the right sub-tree and SI_n contains a null value. On that beat, $P_{1,n}$ receives the root node value which in this case is the next to last value of the inorder enumeration of the whole tree. It puts this value in SI_n and thus at the end of beat $n-1$, SI_n and FI_n contain the last two values of the inorder enumeration of the whole tree. Afterwards, $P_{1,n}$ receives only null values from above and it thus sends only null values to $P_{1,n-1}$. Hence, the arrival of the root node value in $P_{1,n}$ can have no effect on the content of the SI and FI registers of the other base processors and so even after beat $n+j-1$ for any j, $0 < j < n$, the induction hypothesis applies in relation to the left sub-array and processor $P_{1,n-j}$. We conclude that for any j, $1 < j < n$, after beat $(n-1)+j-1$ (i.e. $n+j-2$), SI_{n-j} and FI_{n-j} contain the $(2(n-1-j)-2)^{\mathrm{th}}$ and $(2(n-1-j)-1)^{\mathrm{th}}$ values of the inorder enumeration of the left sub-tree (SI_1 contains a null value). These correspond to the $(2(n-j)-2)^{\mathrm{th}}$ and $(2(n-j)-1)^{\mathrm{th}}$ values of the inorder enumeration of the whole tree and this completes the proof for the case where the right sub-tree consists of only one node. I now consider the case where the right sub-tree consists of more than one node ($1 < m < n$). By the

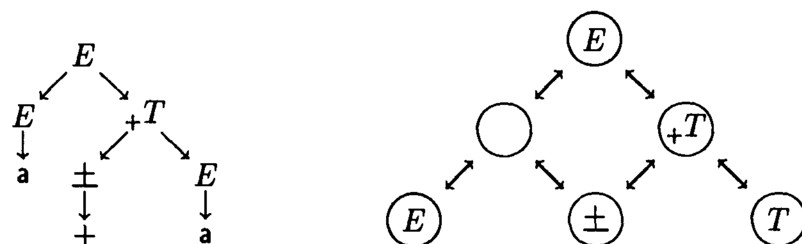
induction hypothesis applied to the right sub-array, when $P_{1,n}$ receives on beat $n-1$ the root node value, it already holds in its SI and FI registers two non-null values. Those are the last two values of the inorder enumeration of the right sub-tree and, consequently, of the whole tree. Because SI_n has a non-null value the root node value is sent to $P_{1,n-1}$ on beat n. By the same argument we can show that the same process as the one just described takes place (propagates) in $P_{1,n-1}$ (if $m > 2$) on beats n and $n+1$ and in general in $P_{1,n-j}$ on beats $n+j-1$ and $n+j$ for any j, $0 < j < m$. That is, on beat $n+j-1$ processor $P_{1,n-j}$ receives from $P_{1,n-j+1}$ the root node value. At that time SI_{n-j} and FI_{n-j} hold the $(2(m-j)-2)^{\mathrm{th}}$ and $(2(m-j)-1)^{\mathrm{th}}$ values of the inorder enumeration of the right sub-tree which are the $(2(n-j)-2)^{\mathrm{th}}$ and $(2(n-j)-1)^{\mathrm{th}}$ values of the inorder enumeration of the whole tree. These values remain there and the root node value is sent to the left on the next beat. On beat $n+m-1$, processor $P_{1,n-m+1}$ receives the root node value. By the induction hypothesis SI_{n-m+1} then contains a null value and FI_{n-m+1} contains the first value of the inorder enumeration of the right sub-tree or equivalently the $(2(n-m)+1)^{\mathrm{th}}$ value of the inorder enumeration of the whole tree. The root node value which is the $2(n-m)^{\mathrm{th}}$ value of this enumeration is stored in SI_{m+1} as required by the theorem. It follows that $P_{1,n-m+1}$ will have sent and will keep sending only null values to $P_{1,m}$ and thus, as in the case where $m = 1$, the induction hypothesis applies integrally to the left sub-array. This completes the proof for the general case. ■

Corolary 8.3.1.1 *When run on an array holding a valid tree representation, algorithm 8.3.1 flattens the tree into its inorder enumeration onto the bottom boundary of the array.*

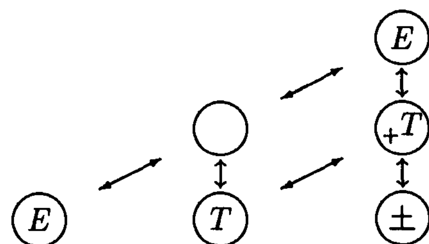
Proof Follows from theorem 8.3.1. ■

8.4 Flattening and the extension to K-GKT

After the marking phase of my extension (section 3.2.2), the K-GKT array holds, like the CIP parse extraction array, a distributed representation of a (parse) tree. In fact, from a K-GKT array holding a tree representation we can obtain the equivalent of a CIP parse extraction array holding the corresponding valid tree representation (satisfying definition 8.1.1) by simple shearing rightward the K-GKT array so as to align vertically processors along the same negative slope diagonals (of the tilted array). Consider, for example, the following parse tree and the corresponding K-GKT array:

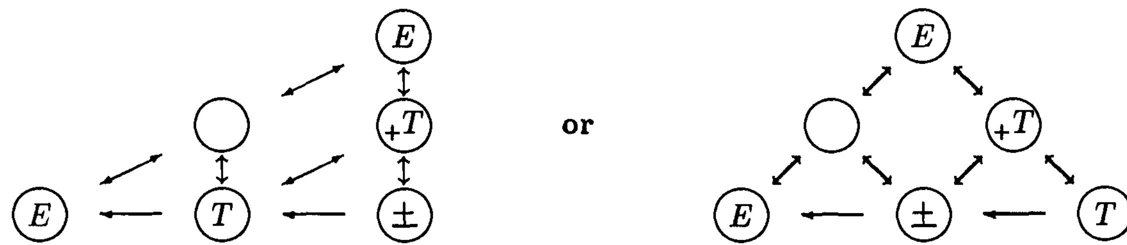


Shearing the array, we get:



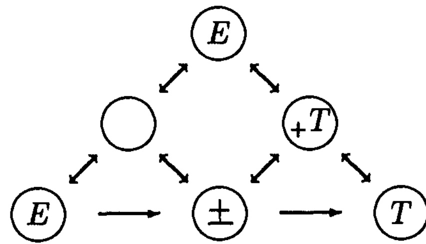
which, if we do not consider communication links, is exactly the same as the CIP parse extraction array corresponding to our tree. This sheared array has all the links needed to run the flattening phase for inorder enumeration (algorithm 8.3.1) and postorder enumeration (the original CIP flattening phase) except for leftward

links to connect the base processors. To run these phases on the K-GKT array, all we need is simply to add these links:



With this modification, these phases can be run on the K-GKT array exactly as on the CIP array.

In the case of the flattening phase for preorder enumeration (algorithm 8.2.1), we must add communication links between processors as well but in the other direction.



Also, we must adapt the algorithm slightly. Algorithm 8.2.1 sends values downward on one beat and then leftward on the next beat. On the K-GKT array we can (and we must) perform this value transfer in one beat instead of two by using the communication links along the positive slope diagonals. The resulting algorithm yields the same results as the original and it terminates (on a $n \times n$ array) $n-1$ beats earlier. We can obtain a formal proof of this by adapting the proof of theorem 8.2.1 as follows: replace all occurrences of "from the right" by "from the top right" and all occurrences of "from $P_{1,2}$" by "from $P_{2,2}$" and add to all expressions containing the term $2n$ the value $n - 1$. I do not deem it worth while or useful to write this proof in full here.

8.4.1 Yet more variants

In section 2.3.6 I described the original CIP flattening phase. In this chapter I introduced two variants of this algorithm and I have just indicated how the three algorithms can all be adapted to run on the K-GKT array. There exist yet other flattening methods of the same pattern. One of them, for instance, consists in gathering the inorder enumeration of a tree represented in a K-GKT array along the top left boundary of the array. When you have seen one variant, you have seen them all. There is no point in going into the details of this last variant or of any of the other variants or even in listing these. The reader may find it more interesting to discover, some of them on his own.

Chapter 9

Discussion

9.1 Comparison with other work

Basically, two algorithms deserve attention in this section: the parsing extension of Chiang and Fu and the parsing algorithm of Chang, Ibarra and Palis (CIP). I have given a brief description of these in sections 2.3.5 and 2.3.6. Both algorithms have been presented by their authors in the restricted framework of CFL parsing but both can easily be generalised and applied to any of the other dynamic programming problems of $\mathcal{C}$.

9.1.1 The extension of Chiang and Fu

Like I, Chiang and Fu have suggested a parsing extension to K-GKT. Instead of implementing the CYK algorithm they chose to implement a weakened version of Earley. They motivated this choice by the fact that CYK requires that we transform a grammar not in CNF into one in CNF. They claim that by resorting to weakened Earley they can avoid the overhead incurred by such a transformation. However, as

I have pointed out, their algorithm implies a preprocessing of the grammar which is the equivalent, but in a more complex form, of the standard transformation of a grammar not in CNF into one in CNF. Moreover, as pointed out by Graham, Harrison and Ruzzo [Graham 76b], weakened Earley is equivalent to CYK. Thus, resorting to weakened Earley provides no real advantage.

The strategy of the Chiang and Fu algorithm consists in accumulating whole parses or sub-parses in each processor. This strategy has two major drawbacks. First, it implies that the amount of information each processor needs to store is proportional to the distance between the processor and the base. As a consequence, the space complexity of the algorithm is $O(n^3)$ in the single parse case. Second, the amount of information exchanged by processors is also proportional to the distance between the processors involved and the base. Hence, the information exchanges occurring at the top of the array take more time than those occurring at its base. Since all the processors of the array need to be synchronized on equal length beats and since the longest information exchanges must be completed in the period between two beats it follows that the beat periods have to be of duration proportional to the input size. For this reason, the time complexity of the algorithm reaches $O(n^2)$.

9.1.2 The CIP algorithm

The CIP algorithm has the same time complexity as mine ($O(n)$) but has a lower space complexity, $O(n^2)$ compared to $O(n^2 \log n)$. Both algorithms use a number of processors proportional to the square of the input length. My algorithm requires exactly $(n+1)n/2$ processors, the version of CIP presented in section 2.3.6 (with two-way inter-processor communication links) requires exactly n^2 processors while the original version [Chang 87] requires exactly $3n^2/2$ processors. The key advantage of CIP over my algorithm is that the size of its processors is totally independent

of the size of the input. Hence if one builds an array of processors to run the CIP algorithm on inputs of size, say, up to 4, one can use exactly the same processors (in greater number) to build an array to handle inputs of any size. In contrast, with my algorithm, the requirement of processor storage space grows in proportion to $\log n$. From a theoretical point of view this will be seen as an important disadvantage of my algorithm over CIP. In practice the disadvantage may be of a lesser significance. Observe that the size of the array being proportional to the square of the input, the storage requirement of each processor grows only in proportion to the log of the square root of the number of processors in the array. Because the $\log n$ factor (above) is due to the counters and pointers used by my algorithm, the hidden constant behind this factor can be relatively small and involve only a "few" bits. The exact number will depend on the grammar used. Another (minor) advantage of CIP over my algorithm is that it is online. The algorithm can run without knowing in advance the size of its input. My algorithm, in contrast, needs to have the whole of the input available before it can start executing.

In its current version CIP can output only one parse of the input. I conjecture that it should be possible to adapt CIP for the production of multiple parses. It would probably be possible to do so by resorting to a strategy like the one I used in the second multiple parse output algorithm presented in section 5.4. This strategy consists in rerunning the algorithm for the production of each parse and in having each run leave a trace in the processor array so as to guide the following run in producing the *next* parse. Whether this strategy can in fact be applied to CIP or not should constitute an interesting subject for further research. Another important topic that warrants further research into the CIP algorithm is the question of decomposability. Can the algorithm be adapted so that it can handle large inputs on arrays that would normally be too small? It is most probably possible to do so simply by having each processor emulate a square array of virtual

processors. But, would an approach similar to the one I used in chapter 6 for my algorithm also provide positive results in the case of CIP?

I must point out that I have obtained most of the results reported in this dissertation before the CIP algorithm came out in January 1987.

9.2 Simple and active memory

My parallel algorithm has the same space complexity ($O(n^2 \log n)$) as its sequential equivalent which in the case of parsing is CYK. A similar comment applies to CIP and the version of CYK that does not use pointers (see [Aho 72] for descriptions of both versions of CYK) if we regard the latter as the sequential equivalent of the former. These two algorithms also have the same space complexity ($O(n^2)$). In other words, in both cases, in term of hardware area requirement, only constant factors separate the parallel algorithms from their sequential counterpart. All four algorithms work with triangular matrices. In the sequential algorithms, the matrix elements are simply memory units which are all serviced by a single processor. In the parallel algorithms the matrix elements are memory units enriched with communication channels and an active circuit unit that we call a processor. The constant factors mentioned above should reflect roughly the differences between the areas occupied by matrix elements consisting of just memory and that occupied by elements consisting of memory, communication channels and a processor.

9.3 Of practical interest ?

In the introduction to this dissertation I have pointed out that technological advances in recent years may justify that we change our views on parallel algorithms which would have been considered of theoretical interest only a decade or so ago. In connection to this it may be appropriate to question the practicality of the algorithm I have presented.

9.3.1 Parsing of programs

As computer scientists, we have been motivated to study parsing primarily because the programs we write have to be parsed. In this context should we consider my algorithm a favorable alternative to the methods currently used? Sadly three reasons strongly suggest a "no" answer to this question. Firstly, computer programs constitute very long problem instances in the context of my algorithm. Very rarely will programs contain less than a thousand tokens and even with that few tokens my algorithm requires more than half a million processors. Notice that such a difficulty can be alleviated via problem decomposition at the expense of execution time. Secondly, almost every programming language is of some restricted type of CFL (LL, LR) for which efficient linear time methods already exist. Thirdly, the advent of language based programming environments [Medina 81] [Reps 83] [Teitelbaum 81] has taken away some of the motivation for faster parsers.

9.3.2 Parsing of natural languages

In natural languages, sentences are rather short compared to computer programs and their length makes them amenable to being parsed using my algorithm. Also an

application like real-time speech recognition for data input [Thompson 84] demands high speed parsing. However, my algorithm would probably not be of much help in this area due to the very serious shortcomings of CFGs in describing natural languages. Another drawback of my algorithm with respect to speech recognition is the fact that it requires to have a whole sentence before it can start parsing it.

9.3.3 Syntactic pattern recognition and pattern matching

Chiang and Fu have suggested the use of CFGs in pattern recognition [Chiang 84] and the K-GKT algorithm can be used to that end. Pattern matching of speech utterances [Tappert 78] is another dynamic programming problem for which the K-GKT algorithm and my extension can be applied. Pattern recognition and pattern matching are two areas where real-time performances are often required. It is not clear however if in these cases recognition (or matching) alone will not always be sufficient or if a parsing capability can also be desirable. The question warrants further investigation.

9.3.4 Building of optimal search trees

The classic textbook example of the application of optimal binary search trees is the search tree for the reserved words in a compiler. One does not build such trees everyday and thus, instead of resorting to a fast algorithm that requires special hardware (e.g. an array of processors) for the task, one might very well prefer to settle for a slow algorithm that runs on the conventional computer at one's disposal. Observe also that the existence of a sequential solution of time complexity $O(n^2)$ for this particular problem makes the parallel solution even less appealing.

The speedup provided by the parallel solution is not optimal in this case but is proportional to the square root of the number of processors.

9.3.5 Optimal matrix multiplication and optimal file merging

Optimal matrix multiplication and optimal file merging are two other problems of $\mathcal{C}$. If we have a series of rectangular matrices to multiply, the order in which we choose to multiply them will determine the total number of scalar multiplication operations that we will need to perform. We will not only want to know how many multiplications the optimal order will involve but will also want to know what the optimal order is. A similar comment applies to the optimal file merging problem. Thus, in these two cases, the partial solution yielded by K-GKT is insufficient and the complete solution, which my extension provides, is required. In practice, instances of these two problems will tend to be short such that one could consider resorting to my algorithm to solve them. On the other hand, in the cases where the instances are very short, the sequential algorithm could prove satisfactory in spite of the fact that its running time is $O(n^3)$. Observe that it is not usually required for problems of this sort to be solved very quickly.

9.3.6 A solution in search of a problem

In the above survey of known problems in $\mathcal{C}$, which is far from exhaustive, I have not identified one practical application for which we could assess a recourse to my algorithms as positively worthwhile. It seems that what we have is a solution in search of a problem and so, in the end, it could well be the case that the research presented in this dissertation should be classified as of theoretical interest. As a consolation (to the reader or to myself?) I offer the following observation. Algo-

rithms that require a number of processors that is proportional to the square (as mine does) or the cube (or...) of the input size are often looked at suspiciously as just playthings for the theoreticians. But consider the following. Suppose that for one problem in $\mathcal{C}$, the basic operation performed during each beat of the K-GKT algorithm required some 6.3 seconds on a powerful processor and suppose that one needed to solve an instance of such a problem of length 1000. Resorting to the uniprocessor approach, one would have to wait over 200 years for the solution. My algorithm would require an array of over half a million processors for an instance of that size. Such an array, if it consisted of powerful processors, would solve the problem in less than 2 hours. If it consisted of not so powerful processors it would solve it within perhaps a few days. The prospect of waiting for 200 years would probably make one envisage to resort to my algorithm in spite the fact that it requires a huge number of processors.

9.4 Areas for further research

9.4.1 Efficient grammars

In chapter 4 on efficient grammars I have mentioned four open questions which make obvious candidates for further research. For convenience to the reader, I echo them here: "how can we characterize efficient grammars?", "is grammar efficiency decidable?", "is *nt*-disjunction of grammars in CNF decidable?" and "is *rhs*-disjunction of grammars in CNF decidable?".

9.4.2 Problem decomposition

In tackling the subject of problem decomposition, I have investigated one scheme which involves using auxiliary storage devices that are connected to processors

on the boundary of the array. Other approaches exist, such as having each real processor emulate a square sub-array of virtual processors. It should be of interest to investigate such other approaches and to find out how they compare with the one I have taken.

9.4.3 CYK combination and PLAs

I have suggested the use of PLAs for the hardware implementation of the CYK combination. I have pointed out that such PLAs, if unoptimised, could be very sparse. It would be interesting to know if we could obtain, by compaction, partitioning or other means, area efficient PLAs for the CYK combination. The efficiency of such PLAs would no doubt be very dependent on the grammar involved. How can we characterize grammars whose CYK combination could be efficiently implemented with PLAs? Could we transform grammars into equivalent grammars that would lead to more efficient CYK combination implementations? Finally, are there other hardware design approaches to the problem that could prove more profitable than PLAs?

9.4.4 Real performance evaluation

Asymptotically, the parallel algorithm presented in this dissertation is a lot more efficient than its sequential counterpart. This implies that even if we had an array whose processors were exceedingly slow, on inputs large enough (assuming the array was of the required size) it would do better than an extremely fast uniprocessor machine. The parallel algorithm no doubt incurs some overhead (amongst other things because it involves inter-processor communications) such that a given uniprocessor machine could be faster in finding the solution to small problem instances than an array whose individual processor performance was comparable to

the performance of the processor in the uniprocessor machine. Experiments would be needed to determine on what size of problem instances a machine like, say, the DAP would start to perform better than some conventional computer and on what size of problem it would perform significantly better. The measurements that we would obtain from such experiments would be very much dependent on the specific problem (of $\mathcal{C}$) we chose for conducting them. For example, CFL parsing involves as the basic operation the CYK combination. The 1 bit processors of the DAP are well suited for executing this operation. On the other hand, they are not so well suited for integer multiplication (in comparison with conventional CPUs), the basic operation in the optimal matrix multiplication problem. It would thus be preferable to experiment with several problems of $\mathcal{C}$. It would also be very interesting to evaluate how the performance obtainable from one type of processor array (e.g. the DAP) compared with the performance obtainable from another (e.g. a Transputer array). Finally, it would be interesting to compare the real term performance of my algorithm with that of CIP.

9.4.5 CIP

As I have mentioned earlier in this chapter (section 9.1.2), it should be possible to adapt the CIP algorithm for the output of multiple parses. Is this in fact true and if so, how exactly can it be done? Finally, it should be interesting to investigate various problem decomposition approaches for CIP.

9.4.6 Transitive closure

As Valiant pointed out [Valiant 75], recognition matrix computation reduces to the problem of boolean matrix transitive closure. Guibas, Kung and Thompson described a systolic algorithm for matrix transitive closure [Guibas 79] (this paper is

the same as the one in which they presented their version of K-GKT). The algorithm computes the transitive closure of an $n \times n$ matrix on an $n \times n$ array of (fixed size) processors in linear time. In the CIP algorithm, a first phase which computes the recognition matrix is followed by a second phase which traces backward the recognition matrix computation (see section 2.3.6). This backward trace allows the recovery of the parse tree. It is perhaps possible to adapt the idea of a backward trace to the Guibas, Kung and Thompson transitive closure algorithm and thus design an extension to this algorithm for parsing. If such is the case, an extension of the sort could well lead to the design of an algorithm that would have the same space and time complexity as CIP. Such an extension would also surely be applicable to the other problems of $\mathcal{C}$.

9.5 Conclusions

In the introduction to this dissertation I have emphasised the fact that much hope is put into the possibilities of profitably exploiting parallelism in computation. I have pointed out the need for a broader knowledge of the promises parallelism can fulfill and of its limitations. With the aspiration of contributing to the enlargement of this knowledge as my main motivation, I have set out to investigate the potential utilisation of parallelism in the field of CFL parsing. I have presented an algorithm (extension) which shows that parallelism can in fact be used for parsing. (When I started my research I was not aware of the existence of the K-GKT algorithm and of its extension by Chiang and Fu. The CIP algorithm was published just after the bulk of my research work had been done.) The recourse to parallelism brings a very significant improvement in the speed at which we can perform CFL parsing (and other problems in $\mathcal{C}$). However, this improvement is obtainable only at a cost which is just as significant. It is not at all obvious whether for this problem, resorting to

parallelism can prove practical or not. The answer to this question will very much depend on the requirements of each specific application. Whatever the case may be, I believe that the research I have presented can be regarded as another brick in the wall of our better knowledge of the possibilities of parallelism.

Appendix A

DAP programs for parsing $L(G_1)$

A.1 Host component

```
*****
*   *
*   *   Program   * * *   A E T K H O S T   * * *
*   *
*****
      PROGRAM AETKHOST
*
*
* AETKHOST is an "upgraded" version of PH2HOST.
*                             See end of this comment.
* PH2HOST is an "upgraded" version of PH1HOST.
*                             See end of this comment.
*
*
*      This program implements on the I.C.L. DAP  distributed Array
*     Processor) the algorithm of Kosaraju and
*     Guibas, Kung and Thompson applied to CFL recognition.
*     The program reads in a string, initialises
*     the relevant array elements and uses the DAP to simulate the
*     systolic algorithm of K-GKT.  To use the DAP, one needs a host
*     program and a DAP program.  This is the HOST one.
*
*     This particular program uses the following grammar:
*
```

```
*          E --> E    +T  |  T
*          T --> T    *F  |  F
*          F --> <(> E)   |  a
*         +T --> <+> T
*         *F --> <*> F
*         E) --> E   <)>
*
*     Grammar dependency is concentrated in the function TOKENSET in
*     the routine SHOWUS and also (unfortunatly) in the choice of
*     integer size of various variables.  This is due to the fact
*     that these integers are used to represent sets of non-terminals.
*
*     The program outputs the non-terminals that derive the input
*     string.
*
*
*  In this upgraded version, we implement the first phase of the
*  parsing algorithm.  This phase is very similar to K-GKT.  The
*  only difference is that we add counters and pointers to be able
*  to reconstruct the parse tree (reconfigure the array).  The
*  only difference this implies for this program is that we added
*  a subroutine (and a call to it) to output the array of pointers
*  (POINTOUT) after recognition.  The DAP program (PH1DAP) is more
*  significantly different than the K-GKT version.  We also modified
*  READIN so that the input string is feeded back on output.
*
*  In this upgraded upgraded version, we now output the "marks" of
*  the processors.  The DAP version (PH2DAP) uses the pointers to
*  mark the processors of the underlying parse tree, either as
*  tree nodes or link nodes.  In this version , we have added a
*  subroutine (and a call to it) to output the array of marks
*  (MARKOUT) after marking.  We keep the subroutine POINTOUT
*  although we do not call it.  Of course, we also call the DAP
*  subroutine that will do the marking after recognition
*  (MARKPROCESSORS).  Well actually, what we do is that we call a
*  subroutine (DAPPHASES) that will call for us the subroutines
*  RECOGNIZE and MARKPROCESSORS.
*
*  You will not find MARKOUT in this file (listing).  Because
*  FORTRAN trapped by software in the subroutine, we had to
*  compile it seperately with a "dont trap" sort of option.
*
*  We have added, for the purpose of better user interface a
*  common block (INPUTSTRING) to output the input string together
*  with the marks of the processors.
*
```

```
*  AETKHOST version: This version is actually a rather lot simpler
*  than the Phase 2 version.  All it does is output the parse and
*  that's pretty simple.  The subroutine PARSOUT takes care of
*  that.  We also needed to modify the PARAMETER value of SIGBIT
*  in the subroutine TOKENSET so as to identify the "base"
*  processors in the array.  We eliminated from this version the
*  subroutine POINTOUT and MARKOUT since they were of no use
*  anymore.  The subroutine PARSOUT is compiled separately to
*  avoid erroneous unassigned variable access software traps by
*  FORTRAN.
*

      INTEGER     NTOKEN, RESULT, PADN1, PADN2, PADN3(124)
      INTEGER*2  ACCSIG(64,64)
      COMMON /ARGMTS/ PADN1, NTOKEN, PADN2, RESULT, PADN3, ACCSIG

*  NTOKEN : number of tokens in the input string (must be <= 64).
*  RESULT : will contain the value of the systolic array root
*           processor.
*  PADN-  : dummy variable to ensure correct allignment with DAP
*           storage mode.
*  ACCSIG : the accumulators and signal controls of the array.
*  ARGMTS : common block to pass arguments to the DAP.

      INTEGER     I
      INTEGER     TOBASE(64), TMDAP, TMDAP1, TMDAP2, TMDAP3, TMDAP4
      DOUBLE PRECISION     TMHOST, TMHOST1, TMHOST2
      CHARACTER  AGAIN

*  I      : loop control variable.
*  TOBASE : contains the values of the systolic array base processors;
*           these are used to initialise the array.
*  TMDAP- : variables to compute the time taken by the DAP.
*  TMHOST-:     "      "     "     "     "     "     "  "  HOST.
*  AGAIN  : to ask the user if he wants to start all over  AGAIN.
*
*   Start of the body of program   * *  A E T K H O S T  * *
*
*
*           Read the input string and initialise the vector TOBASE.
*
   10 CALL READIN (TOBASE,NTOKEN)
*
*   We note here the HOST time to make measures on the execution time.
*
      CALL CPUTIM(TMHOST1)
```

```
*
*   NTOKEN will be zero either if the input string is empty or
*   if it contains illegal characters.
*
      IF (NTOKEN.NE.0) THEN

*          First initialise the whole array to zero.  This is done
*          by a DAP routine.  Before and after, we note the DAP time.

         CALL DAPTIME(TMDAP1)
         CALL INITARRAY
         CALL DAPTIME(TMDAP2)

*          Transfer the content of TOBASE in the processors
*          correspondingto the base of the systolic array.

         DO 20 , I = 1 , NTOKEN
            ACCSIG(NTOKEN+1-I,I) = TOBASE(I)
   20    CONTINUE
*
*          Simulate the systolic algorithm and recognize the string.
*          Via DAP of course.  Here as well, we note DAP times.
*
         CALL DAPTIME(TMDAP3)
         CALL DAPPHASES
         CALL DAPTIME(TMDAP4)
*
*          Output the result in readable form.
*
         CALL SHOWSET(RESULT)
*
*       Output the pointers, that is the counter values saved by
*  each processor.
*
*
*         CALL POINTOUT
*
*   Not anymore.  This is PH2HOST, not PH1HOST.  The DAP has already
*   used the pointer values to mark the nodes of the underlying parse
*   tree.  We willnow output the "marking" of the processors in a
*   readable form.
*
*
*         CALL MARKOUT
*
*   Not anymore.  This is PH3FTN, not PH2HOST.  We now want to output
```

```
*   the parse, if the recognition was successfull.  A successfull
*   recognition is indicated by a strickly positive value for
*   OUTPUTBEATS.  (POINTOUT and MARKOUT do no exist anymore.)  We test
*   a successfull recognition inside the subroutine PARSOUT.
*
        CALL PARSOUT
*
*   Output some execution times.
*
        CALL CPUTIM(TMHOST2)
        TMHOST = TMHOST2 - TMHOST1
        TMDAP  = (TMDAP2 - TMDAP1) + (TMDAP4 - TMDAP3)
        PRINT 2000, TMHOST, TMHOST/NTOKEN,
     X                 TMDAP,  (TMDAP*1.0)/NTOKEN

 2000 FORMAT('  T I M E S            TOTAL          PER TOKEN'//
     X        '           Host : ',         2F15.6/
     X        '           DAP  : ', I8, 7X, F15.6)
      ELSE
*
*   The user has entered either a null string, either an invalid one.
*   Let's ask him if he wants to enter another string or if he's had
*   enough.  The user has to enter 'n' or 'N' to stop the program.
*
        CALL FPROMPT(' Do you want to enter another string? (Y/N)')
        READ  3000 ,  AGAIN
 3000   FORMAT(A1)
        IF ((AGAIN.EQ.'N').OR.(AGAIN.EQ.'n'))   STOP
      ENDIF
*
*   We think this is great fun and the user has not informed us he
*   has a different feeling so we start all over again.
*
      GOTO 10
*
*       END  MAIN PROGRAM   A E T K H O S T
*
      END



*****
*   *
*   *   subroutine    * * *   R E A D I N   * * *
*   *
*****
      SUBROUTINE READIN (TOBASE,NTOKEN)
```

```
*
*    Reads in the input string.  Checks that it's of the right size
*    and that it contains only valid characters.  Returns zero in
*    NTOKEN if any violation.  If not, returns in TOBASE the initial
*    values of the base processors' accumulator and control signals
*    and in NTOKEN, the number of tokens (characters) in the input
*    string.
*
      INTEGER        NTOKEN,  TOBASE(64),  I
      INTEGER*2      TOKENSET
      CHARACTER*80   INSTRING
      LOGICAL        STRINGOK
      COMMON /INPUTSTRING/  INSTRING
*
*  I         : loop control.
*  INSTRING  : the input string.
*  STRINGOK  : string error indicator.
*  TOKENSET  : function computing the set representation associated
*              with a token.
*  INPUT-    : common block to be able to communicate the input
*              string to any interested parties.
*
      STRINGOK = .TRUE.
      CALL FPROMPT('input string please:')
      READ 4000 , INSTRING
 4000 FORMAT(A80)

      NTOKEN = INDEX(INSTRING,' ') - 1
      IF (NTOKEN.EQ.0) THEN
         STRINGOK = .FALSE.
         PRINT * , '***          Input string is null'
      ELSE
         PRINT * , ' The input string is: ', INSTRING
         IF (NTOKEN.GT.64) THEN
            STRINGOK = .FALSE.
            PRINT * , '** ERROR **   Input string too long (>64)'
            NTOKEN = 64
         END IF
      END IF
      DO 100 , I = 1,NTOKEN
         TOBASE(I) = TOKENSET(INSTRING(I:I))
         IF (TOBASE(I).EQ.-1) THEN
            PRINT * , '** ERROR ** Invalid character ''',
     X              INSTRING(I:I),''', in position ',I,'.'
            STRINGOK = .FALSE.
         END IF
```

```
  100 CONTINUE

      IF (.NOT.STRINGOK)    NTOKEN=0
      RETURN
*
*         END  SUBROUTINE  R E A D I N
*
      END



*****
*   *
*   *   function    * * *  T O K E N S E T  * * *
*   *
*****
      INTEGER*2 FUNCTION   TOKENSET  (INCHAR)
*
*    This computes a 16 bit integer whose 10 ls-bits represent the set
*    of non-terminals deriving the character INCHAR and whose 2 next
*    ls-bits (bit 5 and 6) represent the 2 control signals (transfer
*    from the accumulator to the fast belt and transfer from the fast
*    belt to the first stage of the slow belt).  The value computed
*    serves to initialise the base processors.  Since the control
*    signals are sent from the base, bit 5 and 6 are set to 1.
*
*    If the token (character) is illegal, the function returns -1.
*
*      This subroutine has been slightly modified with the additon of
*   counters and pointers in the phase 1 of the parse (PH1HOST).  We
*   now need to add one more control signal (set to one).  This is the
*   signal that will indicate to the processors that their counters
*   have reached their "initial" values.  All that it changes to this
*   subroutine is the PARAMETER value of SIGBIT.
*
*   This is version AETKHOST.  The value of SIGBIT is modified again.
*   We add another bit of value one.  This one will identify the
*   "initialised" processors as the "base" processors of the array.
*
      INTEGER    ASET, LPSET, RPSET, PLUSET, TIMSET, SIGBIT
      CHARACTER  INCHAR

*  ASET   : value representing the set {E,T,F} for the
*           terminal 'a' or 'A'.
*  LPSET  : ... '('.
*  RPSET  : ... ')'.
*  PLUSET : ... '+'.
```

```
*  TIMSET : ... '*'.
*  SIGBIT : value with the four (three(two)) control signals bits at 1.
*           Well, actually, the fourth bit (added in Phase 3) is not a
*           control signal but a mask.
*  INCHAR : the current input character.
*
*    The following PARAMETER values are based on the following
*    convention.
*
*       (MSB is numbeed 1 and LSB, 16, because of the way the
*        EQUIVALENCE matches logical planes of an array to the
*        bits of an integer.)
*
*      non-terminal :  E   T   F   E)   +T   *F   (   )   +   *
*        bit in set :  1   2   3   4    5    6    7   8   9   10
* bit in
*   16 bits integer :  7   8   9   10   11   12   13  14  15  16
*
*    control signal :  stop counter initialisation
* bit in
*   16 bits integer :           4
*
*    control signal :  accum. to fast belt    fast belt to slow belt
* bit in
*   16 bits integer :           5                        6
*
      PARAMETER  (ASET  = 896, LPSET =   8, RPSET =   4,
     X            PLUSET=   2, TIMSET=   1, SIGBIT=15360)
*
*        Compute the set value and initialise the control signal bits
*        while you're at it.
*
          IF ((INCHAR.EQ.'a').OR.(INCHAR.EQ.'A')) THEN
         TOKENSET= SIGBIT + ASET
      ELSE IF (INCHAR.EQ.'(') THEN
         TOKENSET= SIGBIT + LPSET
      ELSE IF (INCHAR.EQ.')') THEN
         TOKENSET= SIGBIT + RPSET
      ELSE IF (INCHAR.EQ.'+') THEN
         TOKENSET= SIGBIT + PLUSET
      ELSE IF (INCHAR.EQ.'*') THEN
         TOKENSET= SIGBIT + TIMSET
      ELSE
         TOKENSET=-1
      END IF
      RETURN
```

```
*
*       END  FUNCTION  ** T O K E N S E T **
*
      END



*****
*   *
*   *   subroutine    * * *   S H O W S E T   * * *
*   *
*****
      SUBROUTINE  SHOWSET  (RESULT)
*
*       This is supposed to show us the result of the computation in
*       a "readable" format.  The way we chose to do that is to put a
*       heading indicating what each bit of the result corresponds to
*       and to write below the heading the values of the bits.
*
      INTEGER        I, RESULT, RSLT, LSBIT, POS
      CHARACTER*80   BITLIN
*
*  I      : loop control variable.
*  RESULT : integer representing the accumulator content and control
*          signals.  We're only interested in the 16 least sign. bits.
*  RSLT   : copy of RESULT.
*  LSBIT  : least significant bit of RSLT.
*  POS    : position where to put the bit representation in string
*           BITLIN.
*  BITLIN : string to contain bit representation with spaces
*           between bits.
*
*       Fill the bit line string with blanks.

      BITLIN = ' '
      RSLT = RESULT
      IF (RSLT.LT.0)THEN
*
*      RSLT is negative.  To extract the bit representation, we must
*      first reverse its bits by adding 1 to it and negating the
*      the result (thus obtaining a positive value).  We then extract
*      the bits of this positive value and invert them.
*
        BITLIN(5:5) = '-'
        RSLT = -(RSLT+1)
        DO 100 , I = 1,15
           POS   = 80 - (I-1)*5
```

```
           LSBIT = MOD(RSLT,2)
           IF (LSBIT.EQ.0) THEN
              BITLIN(POS:POS) = '1'
           ELSE
              BITLIN(POS:POS) = '0'
           END IF
           RSLT = RSLT/2
  100   CONTINUE
      ELSE
*
*       RSLT is positive.  We extract the bit representation directly.
*
        BITLIN(5:5) = '+'
        DO 200 , I = 1,15
           POS   = 80 - (I-1)*5
           LSBIT = MOD(RSLT,2)
           IF (LSBIT.EQ.0) THEN
              BITLIN(POS:POS) = '0'
           ELSE
              BITLIN(POS:POS) = '1'
           END IF
           RSLT = RSLT/2
  200   CONTINUE
      ENDIF

*       Now finally print the results.

      PRINT * , ' The results are:'
      PRINT * ,
      PRINT * , '    -    -    -    - AtoF FtoS    E    T    F   E)',
     X          '    +T   *F    (    )    +    *'
      PRINT * ,
      PRINT * , BITLIN
      RETURN
*
*       END  SUBROUTINE  * *  S H O W S E T  * *
*
      END



*****
*   *
*   *   subroutine    * * *   M A R K O U T   * * *
*   *
*****
      SUBROUTINE  MARKOUT
```

```
*
*  Outputs (in readable form) the relevant portion of the array of
*  processor marks. Processors can be marked either as tree nodes
*  or or as link nodes. Processors can also not be marked at all.
*  Processors linked as link nodes may have been so marked from
*  either their up (vertical) neighbor or their left (horiaontal)
*  neighbor. The same is true for processors marked as tree nodes
*  but we will not need to "depict" this fact. So, all in all, we
*  have to distinguish four different "marks".
*
*  The subroutine is just one big loop (again). Each time the loop
*  is executed, we print the marks of a row of processors. In the
*  first part of the loop, we find the appropriate string
*  representation of the marks and in the second part, we print out
*  these string representations together together with the token
*  associated with the row. Of course, there are NTOKEN rows to
*  consider.
*

      INTEGER      NTOKEN, RESULT, PADN1, PADN2, PADN3(124)
      INTEGER*2    ACCSIG(64,64)
      COMMON /ARGMTS/  PADN1, NTOKEN, PADN2, RESULT, PADN3, ACCSIG

* See main program for a description of the variables.

      CHARACTER*80   INSTRING
      COMMON /INPUTSTRING/ INSTRING

* See subroutine READIN for a description of this variable.

      CHARACTER   CHARMARK(64,64)
      COMMON  /MARKING/  CHARMARK

* CHARMARK   : an array of bytes. Each byte contains various "logical"
*              bits that were used by the DAP during the marking phase.
*              The bits that interest us are only the last three bits.
* MARKING    : a common block to communicate with the DAP.

      INTEGER   INTMARK,   LASTCOLUMN,   I, J
      LOGICAL   VERTICAL,  LINKNODE,  TREENODE
      CHARACTER*2          STRINGMARK(64)

* INTMARK    : the integer representation of a character, used to
*              extract the last three bits
*              (VERTICAL, LINKNODE and TREENODE).
* LASTCOLUMN : the number of processors in a row or in other word, the
```

```
*              column of the last processor of a row.
* I, J       : stupid loop control variables and at the same time smart
*              array indices.
* VERTICAL   : third last bit of a character mark.  Indicates if
               in the case
*            : of a link node, the mark came in the vertical or
*              horizontal direction.
* LINKNODE   : second last bit... .  Indicates if the processor
*              is a link node.
* TREENODE   : last bit... .   ... is a tree node.
* STRINGMARK : array of strings depicting the marks of the processors
*              on a row.

*
*       Start of the body of subroutine  MARKOUT  .
*
*
*    Print a few blank lines.
*
      PRINT * , ' '
      PRINT * , ' '
      DO  200  I = 1 , NTOKEN
*
*  In the first section of this loop, we extract the bit
*  representation of the "mark characters" (of the row considered)
*  and from the last three bits of this representation, we compute a
*  string to depict the mark.
*
*
*        '. '  :  unmarked processor.
*        '| '  :  link node processor marked in the vertical direction.
*        '--'  :    "     "      "         "       "  horizontal    "   .
*        'o '  :  tree node processor.
*
*
*   Compute the number of processors in this row.
*
      LASTCOLUMN =  NTOKEN + 1 - I
*
*   In this inner loop, we consider the marks of each processor in
*   the row in turns.
*
         DO  180  J = 1 , LASTCOLUMN
*
*   Get the integer value of the eight bit character representation.
*
```

```
            INTMARK =  ICHAR (CHARMARK(I,J))
*
*   Extract the last three bits.
*
            TREENODE =  (MOD (INTMARK,2) .EQ. 1)
            INTMARK  =  INTMARK / 2
            LINKNODE =  (MOD(INTMARK,2) .EQ. 1)
            INTMARK  =  INTMARK / 2
            VERTICAL = (MOD(INTMARK,2) .EQ. 1)
*
*   Set the "mark strings" accordingly.
*
            IF (TREENODE) THEN
               STRINGMARK(J) = 'o-'
            ELSE IF (LINKNODE) THEN
                    IF (VERTICAL) THEN
                       STRINGMARK(J) = '| '
                    ELSE
                       STRINGMARK(J) = '--'
                    ENDIF
                 ELSE
                    STRINGMARK(J) = '. '
                 ENDIF

  180    CONTINUE
*
*   Now that we have an array of strings representing the marks of
*   one row of processors, we print them out together with the token
*   associated with this row.
*
      PRINT * , (STRINGMARK(J), J = 1 , LASTCOLUMN), ' ',
     X          INSTRING(LASTCOLUMN:LASTCOLUMN)

  200 CONTINUE
*
*   Print a few blank lines.
*
      PRINT * , ' '
      PRINT * , ' '
*
*   End of subroutine   * * *   M A R K O U T   * * *
*
      END


*****
```

```
*   *
*   *   subroutine    * * *   P A R S O U T   * * *
*   *
*****
      SUBROUTINE   PARSOUT
*
*  This subroutine, added especially for the Phase 3 of the parsing
*  algorithm (PH3FTN), simply outputs the parse that the DAP version
*  is supposed to have stored in the vector PARSE.  For space
*  efficiency reasons, the parse rule numbers are represented in
*  "character" codes so we'll need to do some conversions to recover
*  the rules.  We chose the simplest way to output the parse.  We
*  output the rules one after the other in one column.
*

      INTEGER    NTOKEN, RESULT, PADN1, PADN2, PADN3(124)
      INTEGER*2  ACCSIG(64,64)
      COMMON /ARGMTS/ PADN1, NTOKEN, PADN2, RESULT, PADN3, ACCSIG

* See Main Program for a description of these variables.

      CHARACTER      PARSE(127),  PADCHAR
      INTEGER        PADINT(223), OUTPUTBEATS
      COMMON /PARSING/  PARSE, PADCHAR, PADINT, OUTPUTBEATS

* NTOKEN       : the number of tokens in the input string.
* PARSE        : the vector of rule numbers composing the parse.  The
*                numbers are represented by character codes.
* PADCHAR,     : useless memory space due to Host and DAP storage
*  PADINT        format differences.
* OUTPUTBEATS : some integer that kept a count for us of the number of
*                beats required to get the parse out of the array.
* PARSING      : common block to communicate with the DAP.

      CHARACTER*25   RULES(16)
      INTEGER        I

* RULES        : strings representing the rules as such.
* I            : loop control variable.

      DATA   RULES /  '  (1)  E  -->  E  +T      ',
     X                '  (2)  E  -->  T  *F      ',
     X                '  (3)  E  --> <(>  E)     ',
     X                '  (4)  E  -->           a ',
     X                '  (5) E)  -->  E  <)>     ',
     X                '  (6)  T  -->  T  *F      ',
```

```
    X              '  (7)  T  --> <(>  E)    ',
    X              '  (8)  T  -->           a ',
    X              '  (9) +T  --> <+>  T      ',
    X              ' (10)  F  --> <(>  E)    ',
    X              ' (11)  F  -->           a ',
    X              ' (12) *F  --> <*>  F      ',
    X              ' (13) <(> -->           ( ',
    X              ' (14) <)> -->           ) ',
    X              ' (15) <+> -->           + ',
    X              ' (16) <*> -->           * '   /
*
*   Start of the body of subroutine   * *  P A R S O U T  * *
*
*
*   We output a parse only if recognition was successfull.
*
      IF (OUTPUTBEATS.GT.-1) THEN
         PRINT * , ' '
         PRINT * , ' '
         PRINT * , ' The parse is: '
         PRINT * , ' '
*
*   The parse is twice the length of the input string minus one.
*
         DO 100 ,  I = 1 , 2 * NTOKEN - 1
            PRINT * , ' ', I, '  :      ', RULES(ICHAR(PARSE(I)))

  100    CONTINUE

         PRINT 1000 , OUTPUTBEATS

 1000 FORMAT(/' ',I3,' beats were required to output the parse.'/)

      ELSE
         PRINT * , ' '
         PRINT * , ' *** Recognition failed *** '
         PRINT * , ' '
      ENDIF
      RETURN
*
*   END of subroutine   * *  P A R S O U T  * *
*
      END
```

A.2 DAP component

```
C ********************************************************************
C *                                                                  *
C *                                                                  *
C *                   A  E  T  K   D  A  P                           *
C *                                                                  *
C *                                                                  *
C ********************************************************************
C
C  We've added in a first stage counters and pointers to the original
C  program.
C
C  We are adding in a second stage the marking phase.  This phase uses
C  the pointers (saved counter values) obtained during the recognition
C  to "mark" certain processors as "tree" nodes and certain others as
C  "link" nodes.  The only difference this brings to this program is a
C  (self contained) subroutine MARK_PROCESSORS.  Oh, also, We added a
C  subroutine, DAP_PHASES, that simply acts as an entry point in the
C  DAP program for the Host program.  This subroutine calls the
C  subroutines
C  RECOGNIZE  (not an entry subroutine anymore) and MARK_PROCESSORS.
C
C     PH3DAP :  This version implements the output of the parse (after
C  the marking of the processors of course).  This implementation
C  involves a modification both at the level of phase 1 version and
C  phase 2 version.
C  It also involves pure add-ons.  We need to modify ACC_UPDATE so that
C  it will record the "rule number" (phase 1).  We need to modify MARK_
C  PROCESSORS for two reasons.  First, because our grammar is not
C  "locally
C  unambiguous", local ambiguities need to be resolved during the
C  marking
C  phase.  Second, we identify during this phase the processors at
C  odd and even levels (diagonals) in the array.
C
C     We also modify the common ARGMTS.  Actually, we simply use
C  another bit of the variable SIGACC to identify the base processors.
C  This bit is reset (.FALSE.) by the DAP part (implicitly in
C  INIT_ARRAY) and then set by the Host part.
C
C
C     These subroutines to be executed on the I.C.L. DAP implement
C     the systolic algorithm of Kosaraju
C     and Guibas, Kung and Thompson.  We use this
```

```
C     algorithm to implement in turn the algorithm of Cocke-Younger-
C     Kasami for CFL recognition.
C
C     The grammar used is:
C
C  (1)  E  -->  E  +T       (7)  T  --> <(> E)      (13)  <(>  -->  (
C -(2)  E  -->  T  *F       (8)  T  -->  a          (14)  <)>  -->  )
C -(3)  E  --> <(> E)       (9) +T  -->  <+> T      (15)  <+>  -->  +
C -(4)  E  -->  a          (10)  F  --> <(> E)      (16)  <*>  -->  *
C  (5) E)  -->  E  <)>     (11)  F  -->  a
C  (6)  T  -->  T  *F      (12) *F  --> <*>  F
C
C     Rules 2 has the same right-hand side as rule 6.  During the
C     recognition, rule 2 is recorded and during the marking phase, we
C     determine which of rule 2 or 6 is correct.  Same remark applies
C     for rule 3 (with 7 and 10) and rule 4 (with 8 and 11).
C
C     We number the non-terminal as so:
C
C   non-terminals:     E    T    F   E)   +T   *F   <(>  <)>  <+>  <*>
C        numbers:      1    2    3    4    5    6    7    8    9   10
C
C     We number the terminals as so (numbering not used yet...):
C
C       terminals:     a    (    )    +    *
C         numbers:     1    2    3    4    5
C
C            ****************************************************
C            *                                                  *
C            *     ENTRY SUBROUTINE      I N I T _ A R R A Y    *
C            *                                                  *
C            ****************************************************
C
      ENTRY SUBROUTINE   INIT_ARRAY
C
C Initialises to zero the bits of the matrices of the accumulator
C and the control signals.  These matrices are equivalenced to parts
C of a 16 bits integer matrix ACCSIG and the initialisation is done
C via this matrix.  We resort to this integer matrix for space
C efficiency reasons.
C
C Well finally we decided that this subroutine should also
C  initialise the registers of the fast and slow belts
C
C Now this is not the same program anymore.  We've added things to
C  it on our way to the parser!!  So here we also initialise the
```

```
C  counters and some control bits associated with them.
C
C  This is Phase 3.  In this version, we also initialise the matrix
C  RULE which will be used during the output of the parse.  Of course,
C  we add the declarations of the common block in which RULE appears.
C
      INTEGER  NOFNTERM
      INTEGER     NTOKEN,  RESULT
      INTEGER*2   ACCSIG(,)
      COMMON /ARGMTS/   NTOKEN,  RESULT,  ACCSIG

C   If DAP FORTRAN had the PARAMETER statement:
C
C       LOGICAL  FHBELT(,,NOFNTERM),   FVBELT(,,NOFNTERM),
C                SHBELT(,,NOFNTERM),   SVBELT(,,NOFNTERM),
C                SHBELT2(,,NOFNTERM),   SVBELT2(,,NOFNTERM), TEMP(,)
C
C   But it does not:

      LOGICAL  FHBELT(,,10),   FVBELT(,,10),
     X         SHBELT(,,10),   SVBELT(,,10),
     X         SHBELT2(,,10),   SVBELT2(,,10), TEMP(,)
      COMMON /BELTS/  FHBELT,  FVBELT, SHBELT, SVBELT,
     X                SHBELT2, SVBELT2,        TEMP
      LOGICAL    INIT_COUNT(,),  UPDATE_COUNT(,)
      INTEGER*2  BIG_COUNT(,),   WEE_COUNT(,),
     X           V_POINTER(,),   H_POINTER(,)
      COMMON /COUNTING/  INIT_COUNT, UPDATE_COUNT,
     X                   BIG_COUNT,  WEE_COUNT,
     X                   V_POINTER,  H_POINTER

C See subroutine RECOGNIZE for descriptions of these variables.

      INTEGER*1     LEFT_SIDE(,),  RIGHT_SIDE_1(,),  RIGHT_SIDE_2(,),
     X              RULE(,)
      LOGICAL       ODD_LEVEL(,)
      COMMON  /RULING/  LEFT_SIDE, RIGHT_SIDE_1, RIGHT_SIDE_2,
     X                  RULE,      ODD_LEVEL

C See subroutine ACC_UPDATE for a description of these variables.

      INTEGER   I

C I  :  Loop control variable.
C
C   We set the value of "PARAMETER" NOFNTERM, the number of non-
```

```
C   terminals in the grammar.
C
      NOFNTERM = 10
C
C        We do not convert from 2900 FORTRAN to DAP FORTRAN, since we
C        initialise.
C
      ACCSIG(,)  = 0
      DO 50   I = 1, NOFNTERM
        FVBELT (,,I) = .FALSE.
        FHBELT (,,I) = .FALSE.
        SHBELT (,,I) = .FALSE.
        SVBELT (,,I) = .FALSE.
        SHBELT2(,,I) = .FALSE.
        SVBELT2(,,I) = .FALSE.
   50 CONTINUE
      INIT_COUNT    = .TRUE.
      UPDATE_COUNT  = .FALSE.
      BIG_COUNT     = 1
      WEE_COUNT     = 0
C
C   It sure is not necessary to initialise V_POINTER and H_POINTER
C   but it might come in handy if we want to see  what's  going on...
C
      V_POINTER     = -1
      H_POINTER     = -1
C
C   We don't even need to convert from DAP to 2900 since all the
C   bits are at zero.  Maybe I must add here that only ACCSIG
C   will be used (modified) anyway by the host.
C
C
C   Added bit for phase 3.
C
      RULE = 0
      RETURN
C
C        END OF ENTRY SUBROUTINE  * *  I N I T _ A R R A Y  * *
C
      END


C           *****************************************************
C           *                                                   *
C           *         SUBROUTINE        R E C O G N I Z E       *
C           *                                                   *
```

```
C           ******************************************************

      SUBROUTINE   RECOGNIZE
C
C       Does the bulk of the work really.  In other words, this is
C    IT!!  The actual implementation of the K-GKT algorithm.  All the
C    rest was simply initialisations, inputs and outputs.  On the
C    other hand, we should not get too excited here.  This subroutine
C    is only a big loop.  In this loop we do the following things:
C
C       - Transfer the accumulator contents (of selected
C         accumulators) to fast belt registers.
C       - Transfer data in-between processors.
C       * Increment (half of the) counters that have not yet reached
C         their initial value.
C      *- Transfer the control signals.
C       - Transfer fast belt register contents (of selected
C         fb registers) to slow belt registers.
C      *- Update content of all accumulators using the values
C         in the fast and slow belt registers.
C       * Update the counters of those processors currently computing
C         their value.
C
C    The interesting thing about this stupid looking loop is that we
C    could start executing it from anywhere within it, provided the
C    initialisations are in accordance.
C
C    (-)  steps present in the initial program (pure K-GKT).
C    (*)  steps added in the second program    (phase 1).
C   (*-)  steps present in the first program and modified in the
C         second.
C
C  This is the Phase 3 version.  Here, we simply add a variable,
C BASE_NODE.  Note that Phase 3 involves changes in ACC_UPDATE and
C that C ACC_UPDATE is called by this subroutine.
C After some thinking, we have found out that we need also to
C initialise the rules for the base nodes in this subroutine.
C That means we also need to import a common (RULING).
C
      INTEGER    NOFNTERM
      INTEGER    NTOKEN,  RESULT
      INTEGER*2  ACCSIG(,)

C    If DAP FORTRAN had the PARAMETER statement:
C
C       LOGICAL    SIGACC(,,16),
```

```
C                INIT_C_STOP(,), ACCTOFB(,), FBTOSB(,),
C                ACCUM(,,NOFNTERM)
C
C   But it does not so:

      LOGICAL    SIGACC(,,16),
     X           INIT_C_STOP(,), ACCTOFB(,), FBTOSB(,),
     X           ACCUM(,,10)
C
C   Added for Phase 3.  We EQUIVALENCE this bit (BASE_NODE) to a bit of
C   SIGACC.
C
      LOGICAL    BASE_NODE(,)
      EQUIVALENCE (SIGACC,ACCSIG),
     X            (SIGACC(,,3),BASE_NODE),
     X            (SIGACC(,,4),INIT_C_STOP),
     X            (SIGACC(,,5),ACCTOFB), (SIGACC(,,6),FBTOSB),
     X            (SIGACC(,,7),ACCUM)
      COMMON /ARGMTS/   NTOKEN, RESULT, ACCSIG

C NOFNTERM  :  Number of non-terminals in the grammar.
C NTOKEN    :  Number of tokens in the input string (must be <= 64).
C RESULT    :  The result of the recognition (set of processor (1,1)).
C ACCSIG    :  Integer representation of the ACCumulator registers and
C              the control signals.
C SIGACC    :  EQUIVALENCED logical (bit) representation of ACCSIG.
C INIT_C_STOP: Control signal indicating to a processor that
C              its Counters have reached their "INITial" values
C              and that it can STOP initialising (incrementing) them.
C ACCTOFB   :  ACCumulator TO Fast Belt control signals.
C FBTOSB    :  Fast Belt TO Slow Belt control signals.
C ACCUM     :  ACCUMulator registers.
C BASE_NODE :  indicates that the processor is at the "base" of the
C              array.  By "base", we mean the diagonal at a distance
C              NTOKEN from the "root" (the top left corner).  (Added
C              in the Phase 3 version.)

      LOGICAL    INIT_COUNT(,),  UPDATE_COUNT(,)
      INTEGER*2  BIG_COUNT(,),   WEE_COUNT(,),
     X           V_POINTER(,),   H_POINTER(,)
      COMMON /COUNTING/  INIT_COUNT, UPDATE_COUNT,
     X                   BIG_COUNT,  WEE_COUNT,
     X                   V_POINTER,  H_POINTER

C INIT_COUNT   : Indicates to a processor that he is presently in
C                the process of initialising his counters (during
```

```
C                 the recognition phase).
C UPDATE_COUNT : Indicates to a processor that it is presently in the
C                 process of computing its value and that at every beat,
C                 it must update its counters.  Both INIT_COUNT and
C                 UPDATE_COUNT are "masks" that serve to select the
C                 processors that will be involved in counter value
C                 modifications.
C BIG_COUNT    : The counter that will hold the bigger values.
C WEE_COUNT    : The one ... smaller ....
C V_POINTER    : Saved counter value pointing to a previous processor
C                 in the Vertical direction.  This processor is the one
C                 that will have computed the value responsible for the
C                 insertion of a value in the processor's set.
C H_POINTER    : Saved ... Horizontal direc... .

      LOGICAL  FHBELT(,,10),  FVBELT(,,10),
     X         SHBELT(,,10),  SVBELT(,,10),
     X         SHBELT2(,,10),  SVBELT2(,,10), TEMP(,)
      COMMON /BELTS/  FHBELT,  FVBELT, SHBELT, SVBELT,
     X                SHBELT2, SVBELT2,        TEMP

C FHBELT   :  Fast Horizontal BELT register.
C FVBELT...
C SHBELT...
C SVBELT...
C SHBELT2  :  Second Slow Horizontal BELT register, the one that
C        slows things down.
C SVBELT2...
C TEMP     :  Temporary variable (bit plane) for register transfers.

      INTEGER*2   AF_STATE,    FS_STATE
      LOGICAL     AF_ACTIVE,   FS_ACTIVE,   BEAT_IS_EVEN
      COMMON /SIG_STATUS/  AF_STATE,  FS_STATE,
     X                     AF_ACTIVE, FS_ACTIVE, BEAT_IS_EVEN

C BEAT_IS_EVEN : Indicates whether we are on an even beat or not.
C                 We toggle this variable between .TRUE. and .FALSE..

C AF_STATE :  See subroutine UPDATESIGNALS for a description of the
C             other SIG_STATUS variables.
C
C   Added for Phase 3:
C
      INTEGER*1    LEFT_SIDE(,),  RIGHT_SIDE_1(,),  RIGHT_SIDE_2(,),
     X             RULE(,)
      LOGICAL      ODD_LEVEL(,)
```

```
      COMMON  /RULING/  LEFT_SIDE, RIGHT_SIDE_1, RIGHT_SIDE_2,
     X                  RULE,      ODD_LEVEL

C See subroutine ACC_UPDATE for a description of these variables.

      INTEGER   NOFBEATS
      INTEGER   I, J

C NOFBEATS :  Number of times we do the big loop which is equivalent to
C             the number of beats the simulated systolic array has to
C             go through to compute the sought result.
C I, J     :  Loop control variables.
C
C   We don't need the following since "plane geometry" is on
C   by default, but just in case...
C


      GEOMETRY(PLANE,PLANE)
C
C   We set the value of "PARAMETER" NOFNTERM, the number of non-
C   terminals in the grammar.
C
      NOFNTERM = 10
C
C   We initialise the signal status global variables.
C   We start with a transfer from the accumulators along
C   the base to the fast belts.  Soon after, FS_STATE will
C   turn to zero and we will transfer from fast belt to
C   slow belt on the processor above the base.
C
      AF_STATE  =  0
      FS_STATE  =  2
      AF_ACTIVE = .TRUE.
      FS_ACTIVE = .FALSE.
C
C   The first beat, beat 1, is odd, isn't it?
C
      BEAT_IS_EVEN = .FALSE.
C
C   We never needed to do 2900/DAP conversions before but this
C   time, we can not avoid it.  The host (2900) modified certain
C   values of ACCSIG so we need to convert it.
C
      CALL CONVFM2 (ACCSIG)
C
C   PH3DAP:  Bit added for the Phase 3 version:
```

```
C
C   We record here the rules used to initialise the "base node"
C   processors accumulators.
C
C   recall:  non-terminal    E   <(>   <)>   <+>   <*>
C                  number    1    7     8     9    10
C
C  rule(4)   E  -->   a

      RULE(BASE_NODE .AND. ACCUM(,,1)) =  4

C  rule(13) <(> -->   (

      RULE(BASE_NODE .AND. ACCUM(,,7)) =  13

C  rule(14) <)> -->   )

      RULE(BASE_NODE .AND. ACCUM(,,8)) =  14

C rule(15)  <+> -->   +

      RULE(BASE_NODE .AND. ACCUM(,,9)) =  15

C rule(16)  <*> -->   *

      RULE(BASE_NODE .AND. ACCUM(,,10)) =  16
C
C   Here is where the big loop starts.  We loop around
C   NTOKEN*2 - 3) times.  Usually, we would loop NTOKEN*2 times but
C   because of the way we have set up the loop and the initialisa-
C   tions, we can save 3 beats.
C
      NOFBEATS = NTOKEN * 2  -  3
C
C     FORTRAN does not allow loop parameters to be less than zero
C     so we have to do this little test here.
C
      IF (NOFBEATS.LT.1) GOTO 101
      DO 100   I = 1  ,  NOFBEATS
C
C   Transfer the content of the accumulator of selected
C   processors to the processors fast belt registers.  Now this
C   is not done all the time.  In fact, half of the time its
C   not done.
C
      IF (.NOT.AF_ACTIVE) GOTO 11
```

```
        DO 10   J = 1 , NOFNTERM
        FHBELT(ACCTOFB,J) = ACCUM(,,J)
        FVBELT(ACCTOFB,J) = ACCUM(,,J)
   10   CONTINUE
   11 CONTINUE
C
C   Execute the inter-processor data transfers.
C
      DO 20   J = 1, NOFNTERM

C   Horizontaly first.
C         The root is processor (1,1).  Can you
C         see what this implies?
C            Our array is tilted 90 degrees counteclockwise
C            compared to the array GKT in their Caltech 79
C            paper.  So our horizontal belts correspond to
C            their vertical belts and...

        FHBELT(,,J) = FHBELT(,+,J)
        TEMP(,)     = SHBELT(,,J)
        SHBELT(,,J) = SHBELT2(,+,J)
        SHBELT2(,,J)= TEMP(,)

C   Vertically now.
C         Look where the "shift indices" are now.

        FVBELT(,,J) = FVBELT(+,,J)
        TEMP(,)     = SVBELT(,,J)
        SVBELT(,,J) = SVBELT2(+,,J)
        SVBELT2(,,J)= TEMP(,)
   20 CONTINUE
C
C   Increment either the BIG counter, either the WEE one, depending
C   on which turn it is.
C
      IF (      BEAT_IS_EVEN) BIG_COUNT(INIT_COUNT) = BIG_COUNT + 1
      IF (.NOT. BEAT_IS_EVEN) WEE_COUNT(INIT_COUNT) = WEE_COUNT + 1
C
C   Update the control signals as need be.
C
      CALL UPDATESIGNALS
C
C   Copy content of Fast Belt register to the first slow belt
C   registers.
C
      IF (.NOT.FS_ACTIVE) GOTO 91
```

```
        DO 90   J = 1, NOFNTERM
        SHBELT(FBTOSB,J) = FHBELT(,,J)
        SVBELT(FBTOSB,J) = FVBELT(,,J)
   90   CONTINUE
   91 CONTINUE
C
C   Update the content of the accumulators.
C
      CALL ACC_UPDATE
C
C   Now modify the counters of those processors currently computing
C   their values.
C
      BIG_COUNT(UPDATE_COUNT) = BIG_COUNT + 1
      WEE_COUNT(UPDATE_COUNT) = WEE_COUNT - 1
C
C   Allright, we're almost finished with this loop.  One last thing we
C   must do before the end is to toggle the value of BEAT_IS_EVEN.
C
      BEAT_IS_EVEN =  .NOT.  BEAT_IS_EVEN
C
C   That's the end of this loop.
C
  100 CONTINUE

  101 CONTINUE
C
C   It's almost finished now.  What we are looking for is in
C   the accumulator of processor (1,1).  The only thing we
C   need to do now is to put this value in a scalar variable
C   so as to be available to the 2900.  Resorting to a
C   separate variable allows us to avoid having to convert
C   the whole accumulator matrix.
C
       RESULT = ACCSIG (1,1)

C
C   Well just up there is some obsolete stuff.  We are now not
C   looking only for the value of processor (1,1).  We are
C   interested in the value of all the processors and most of all
C   in the values of the pointers: V_POINTER and H_POINTER.  So
C   we'll convert the relevant matrix just now and the host
C   program will sort out how to present them to you dear user.
C
C   All this up there was true in PH1DAP but is not in PH2DAP.
C   Now we want the DAP to use the pointer values so we'll not
```

```
C   convert anything but instead, we'll put the following lines
C   in comments.
C
C
C      CALL CONVMF2 (ACCSIG)
C      CALL CONVMF2 (V_POINTER)
C      CALL CONVMF2 (H_POINTER)
C
C
C   Just in case things go badly, we'll convert the counter matrices
C   as well so we can look at their content if we want to.
C
C
C      CALL CONVMF2 (BIG_COUNT)
C      CALL CONVMF2 (WEE_COUNT)

      RETURN
C
C   END OF SUBROUTINE  * *   R E C O G N I Z E   * *
C
      END


C          ******************************************************
C          *                                                    *
C          *    SUBROUTINE        A C C _ U P D A T E           *
C          *                                                    *
C          ******************************************************
      SUBROUTINE  ACC_UPDATE
C
C      This procedure updates the content of the accumulators
C    accordingly with the values on the fast and slow belts.
C    The procedure is completely dependant on the grammar used.
C
C      Let's repeat here the numbering convention of the non-
C    terminals so as to make it more convenient to read this
C    seemingly meaningless code.
C
C        non-terminal  :  E  T  F  E)  +T  *F  (  )  +  *
C        bit number    :  1  2  3  4   5   6   7  8  9 10
C
C      To minimise the computations, we update by applying the
C    rules in a given order.  Can you see why?  In fact, we use
C    single production rules (NT --> NT) although this does not
C    comply with the Chomski Normal Form.
C
```

```
C       This is not the original ACC_UPDATE.  It's the version in
C    which we have the counters and we save them once in a while.
C    To celebrate this occasion, we have introduced two new logical
C    DAP matrix variables.  Their declaration follow.
C
C       Well, they follow this added lines of comment.  This ACC_
C    UPDATE is now the one for phase 3.  What we do that is new is
C    that we record the rules that allow us to make insertions in the
C    set of non-terminals.  Actually, because our grammar is locally
C    ambiguous, we record a rule that allows an insertion but at the
C    time of recording, we are not sure yet if this rule is going to
C    be the one we are looking for.  This uncertainty will be resolved
C    in the marking phase.
C
C       We record the rules at the end of the subroutine.
C
C
      LOGICAL    PAIR_FV_SH(,),  PAIR_SV_FH(,)

C PAIR_FV_SH : Indicate if the processor had on its Fast Vertical belt
C              and its Slow Horizontal belt a pair of value
C              corresponding to a right hand side rule.  This is going
C              to be used to mask out processors when assigning values
C              to the pointers V_POINTER and H_POINTER.
C PAIR_SV_FH : Indicates if ... Slow Vertical ... Fast... .

      INTEGER    NTOKEN,  RESULT
      INTEGER*2  ACCSIG(,)

C    If DAP FORTRAN had the PARAMETER statement:
C
C       LOGICAL     SIGACC(,,16),
C                   INIT_C_STOP(,), ACCTOFB(,), FBTOSB(,),
C                   ACCUM(,,NOFNTERM)
C
C    But it does not so:

      LOGICAL    SIGACC(,,16),
     X           INIT_C_STOP(,), ACCTOFB(,), FBTOSB(,),
     X           ACCUM(,,10)
      LOGICAL    BASE_NODE(,)
      EQUIVALENCE (SIGACC,ACCSIG),
     X            (SIGACC(,,3),BASE_NODE),
     X            (SIGACC(,,4),INIT_C_STOP),
     X            (SIGACC(,,5),ACCTOFB), (SIGACC(,,6),FBTOSB),
     X            (SIGACC(,,7),ACCUM)
```

```
      COMMON /ARGMTS/   NTOKEN, RESULT, ACCSIG

C   If DAP FORTRAN had the PARAMETER statement:
C
C      LOGICAL  FHBELT(,,NOFNTERM),   FVBELT(,,NOFNTERM),
C     X         SHBELT(,,NOFNTERM),   SVBELT(,,NOFNTERM),
C     X         SHBELT2(,,NOFNTERM),  SVBELT2(,,NOFNTERM),  TEMP(,)
C
C   But it does not:

      LOGICAL  FHBELT(,,10),   FVBELT(,,10),
     X         SHBELT(,,10),   SVBELT(,,10),
     X         SHBELT2(,,10),   SVBELT2(,,10), TEMP(,)
      COMMON /BELTS/  FHBELT,  FVBELT, SHBELT, SVBELT,
     X                SHBELT2, SVBELT2,        TEMP
      LOGICAL    INIT_COUNT(,),  UPDATE_COUNT(,)
      INTEGER*2  BIG_COUNT(,),   WEE_COUNT(,),
     X           V_POINTER(,),   H_POINTER(,)
      COMMON /COUNTING/  INIT_COUNT, UPDATE_COUNT,
     X                   BIG_COUNT,  WEE_COUNT,
     X                   V_POINTER,  H_POINTER

C  For variable descriptions, see subroutine RECOGNIZE.
C
C   The next variables were added for phase 3.
C
      INTEGER*1    LEFT_SIDE(,),  RIGHT_SIDE_1(,),  RIGHT_SIDE_2(,),
     X             RULE(,)
      LOGICAL      ODD_LEVEL(,)
      COMMON  /RULING/  LEFT_SIDE, RIGHT_SIDE_1, RIGHT_SIDE_2,
     X                  RULE,      ODD_LEVEL

C LEFT_SIDE    : Non-terminal number of this processor.
C RIGHT_SIDE_1 : First non-terminal of the (common) right hand side of
C        the (possibly many) rule(s) by which we inserted non-
C        terminals in the set of this processor.
C RIGHT_SIDE_2 ...
C RULE         : Before the marking phase, any rule whose right hand
C        side allowed the insertion of non-terminals in the set of the
C        processor.  After the marking, the specific rule for this
C        non-terminal, i.e. the one with a right-hand side as above
C        and with a left-hand side corresponding to the lhs of the
C        processor.
C ODD_LEVEL    : Indicate the parity of the level a processor is on.
C        A level is simply a diagonal of the array (from bottom-
C        left to top-right).  Processor (1,1) constitutes level 1,
```

```
C       the diagonal "below" level two and so on.
C RULING       : A common block just to be able to share these
C       variables among various DAP FORTRAN only subroutines.
C
C   Observe on the following statements that we always look
C   for the first non-terminal of a right-hand side on the
C   vertical belts and for the second on the horizontal belts.
C   This is contrary to the convention found in the literature on
C   K-GKT.  The reason is that also contrary to the convention, we
C   put the "root" of the array at processor (1,1), the North-West
C   corner.  Hence, the base
C   lies on a diagonal (perpendicular to "conventionnal" diagonals)
C   touching the west and north boundaries of the DAP array.  So
C   vertical belts carry information pertaining to left substrings
C   of the input and horizontal ... right ...
C
C
C   We don't need the following since "plane geometry" is on
C   by default but just in case...
C
      GEOMETRY(PLANE,PLANE)
C
C   *F  -->  *  F
C
      ACCUM(,,6) = ACCUM(,,6).OR.(FVBELT(,,10).AND.SHBELT(,,3))
     X                        .OR.(SVBELT(,,10).AND.FHBELT(,,3))
C
C   +T  -->  +  T
C
      ACCUM(,,5) = ACCUM(,,5).OR.(FVBELT(,,9).AND.SHBELT(,,2))
     X                        .OR.(SVBELT(,,9).AND.FHBELT(,,2))
C
C   E)  -->  E  )
C
      ACCUM(,,4) = ACCUM(,,4).OR.(FVBELT(,,1).AND.SHBELT(,,8))
     X                        .OR.(SVBELT(,,1).AND.FHBELT(,,8))
C
C   F  -->  (  E)
C
      ACCUM(,,3) = ACCUM(,,3).OR.(FVBELT(,,7).AND.SHBELT(,,4))
     X                        .OR.(SVBELT(,,7).AND.FHBELT(,,4))
C
C   T  -->  T  *F   |   (  E)
C
C   Actually, we use the following rule:
C
```

```
C       T  -->  T  *F    |   F
C
C   which because of the order of evaluation of the other rule
C   is equivalent.
C
      ACCUM(,,2) = ACCUM(,,2).OR.(FVBELT(,,2).AND.SHBELT(,,6))
     X                        .OR.(SVBELT(,,2).AND.FHBELT(,,6))
     X                        .OR.ACCUM(,,3)
C
C   E  -->  E  +T   |    T  *F   |   ( E)
C
C   Actually, we use the equivalent rule:
C
C       E  -->  E  +T   |   T
C
      ACCUM(,,1) = ACCUM(,,1).OR.(FVBELT(,,1).AND.SHBELT(,,5))
     X                        .OR.(SVBELT(,,1).AND.FHBELT(,,5))
     X                        .OR.ACCUM(,,2)
C
C That used to be all we did in this subroutine but in this version,
C we implement the counter and pointer idea and we need to have the
C lines that follow to do that.  If a pair of value of the belts
C correspond to a right hand side we save the counters in the pointer
C variables (V_POINTER and H_POINTER).  The fast belts always carry
C values from nearer processors (those whose distance are indicated by
C WEE_COUNT and the slow... .  Can you see how this fact is reflected
C in the following code?
C
      PAIR_FV_SH = (FVBELT(,,10).AND.SHBELT(,,3)).OR.
     X             (FVBELT(,, 9).AND.SHBELT(,,2)).OR.
     X             (FVBELT(,, 1).AND.SHBELT(,,8)).OR.
     X             (FVBELT(,, 7).AND.SHBELT(,,4)).OR.
     X             (FVBELT(,, 2).AND.SHBELT(,,6)).OR.
     X             (FVBELT(,, 1).AND.SHBELT(,,5))
      PAIR_SV_FH = (SVBELT(,,10).AND.FHBELT(,,3)).OR.
     X             (SVBELT(,, 9).AND.FHBELT(,,2)).OR.
     X             (SVBELT(,, 1).AND.FHBELT(,,8)).OR.
     X             (SVBELT(,, 7).AND.FHBELT(,,4)).OR.
     X             (SVBELT(,, 2).AND.FHBELT(,,6)).OR.
     X             (SVBELT(,, 1).AND.FHBELT(,,5))
      V_POINTER (PAIR_FV_SH) = WEE_COUNT
      V_POINTER (PAIR_SV_FH) = BIG_COUNT
      H_POINTER (PAIR_FV_SH) = BIG_COUNT
      H_POINTER (PAIR_SV_FH) = WEE_COUNT
C
C And in this version (PH3DAP), we also implement the output of the
```

```
C parse.  So we now need to record the rules that allowed the
C insertions.  (see comment at the beginning of the subroutine).
C
C  *F  -->  *  F

      RULE(    (FVBELT(,,10).AND.SHBELT(,,3))
     X      .OR.(SVBELT(,,10).AND.FHBELT(,,3))) =  12

C  +T  -->  +  T

      RULE(    (FVBELT(,, 9).AND.SHBELT(,,2))
     X      .OR.(SVBELT(,, 9).AND.FHBELT(,,2))) =   9

C  E)  -->  E  )

      RULE(    (FVBELT(,, 1).AND.SHBELT(,,8))
     X      .OR.(SVBELT(,, 1).AND.FHBELT(,,8))) =   5

C   E,T,F  -->  (  E)
C             { "(  E)" is a right-hand side common to E,T,F }

      RULE(    (FVBELT(,, 7).AND.SHBELT(,,4))
     X      .OR.(SVBELT(,, 7).AND.FHBELT(,,4))) =   3

C   E,T    -->  T  *F        { "T  *F" is a ... to E,T }

      RULE(    (FVBELT(,, 2).AND.SHBELT(,,6))
     X      .OR.(SVBELT(,, 2).AND.FHBELT(,,6))) =   2

C   E  -->  E  +T

      RULE(    (FVBELT(,, 1).AND.SHBELT(,,5))
     X      .OR.(SVBELT(,, 1).AND.FHBELT(,,5))) =   1
      RETURN
C
C   END SUBROUTINE   * *  A C C _ U P D A T E  * *
C
      END


C            ****************************************************
C            *                                                  *
C            *    SUBROUTINE    U P D A T E _ S I G N A L S     *
C            *                                                  *
C            ****************************************************
```

```
      SUBROUTINE   UPDATE_SIGNALS
C
C      The control signals cycle through states.  In a given
C   state, they are activated and in given states they are
C   transfered to neighboring processors.  The states are held in
C   two global variables AF_STATE and FS_STATE (possible because
C   our machine is SIMD).  Two global logical variables, AF_ACTIVE
C   and FS_ACTIVE, indicate if the signals are active.
C
C      Actually, this whole state business is just a way to simulate
C   signals that travel at certain speeds.  The Accumulator to Fast
C   belt signals travel at the speed of the slow belt and the other
C   ones travel at 2/3 of that speed.
C
C      This subroutine updates the states of the control signals and
C   their activation indicator accordingly.  If need be, it executes
C   control signal transfers.
C
C
C      Like the other subroutines, this one is also modified for the
C   implementation with added counters.  Its in this subroutine that
C   we transfer the signals that indicate to the processors when their
C   counters have reached their "initial" value, when to start
C   modifying them and when to stop modifying them.  For the "initial"
C   bit, we have an extra control signal (INIT_C_STOP) and something
C   that remember its passage (INIT_COUNT) while for the modifying
C   bit, we use the transfer control signals we already had and also
C   something to "remember" (UPDATE_COUNT)...
C
      INTEGER*2   AF_STATE,    FS_STATE
      LOGICAL     AF_ACTIVE,   FS_ACTIVE,    BEAT_IS_EVEN
      COMMON /SIG_STATUS/  AF_STATE,  FS_STATE,
     X                     AF_ACTIVE, FS_ACTIVE, BEAT_IS_EVEN

C  For descriptions of the above variables see above comment
C  (except for BEAT_IS_EVEN, see subroutine RECOGNIZE).

      INTEGER     NTOKEN,  RESULT
      INTEGER*2   ACCSIG(,)

C   If DAP FORTRAN had the PARAMETER statement:
C
C      LOGICAL     SIGACC(,,16),
C                  INIT_C_STOP(,), ACCTOFB(,), FBTOSB(,),
C                  ACCUM(,,NOFNTERM)
C
```

```
C   But it does not so:

      LOGICAL    SIGACC(,,16),
     X           INIT_C_STOP(,), ACCTOFB(,), FBTOSB(,),
     X           ACCUM(,,10)
      LOGICAL    BASE_NODE(,)
      EQUIVALENCE (SIGACC,ACCSIG),
     X            (SIGACC(,,3),BASE_NODE),
     X            (SIGACC(,,4),INIT_C_STOP),
     X            (SIGACC(,,5),ACCTOFB), (SIGACC(,,6),FBTOSB),
     X            (SIGACC(,,7),ACCUM)
      COMMON /ARGMTS/   NTOKEN, RESULT, ACCSIG
      LOGICAL    INIT_COUNT(,),  UPDATE_COUNT(,)
      INTEGER*2  BIG_COUNT(,),   WEE_COUNT(,),
     X           V_POINTER(,),   H_POINTER(,)
      COMMON /COUNTING/  INIT_COUNT, UPDATE_COUNT,
     X                   BIG_COUNT,  WEE_COUNT,
     X                   V_POINTER,  H_POINTER

C  For descriptions of these, see subroutine RECOGNIZE.
C
C   We don't need the following since "plane geometry" is on
C   by default, but just in case...
C
      GEOMETRY(PLANE,PLANE)
C
C   We set the value of "PARAMETER" NOFNTERM, the number of non-
C   terminals in the grammar.
C
      NOFNTERM = 10

C   Accumulator to Fast belt control signal update first
C   (order not important).

      IF (AF_STATE.EQ.1)   ACCTOFB(,) = ACCTOFB(,+)
      AF_STATE = AF_STATE + 1
      IF (AF_STATE.GT.1)   AF_STATE = 0
      AF_ACTIVE = (AF_STATE.EQ.0)

C   Now the Fast to Slow belt...

      IF ((FS_STATE.EQ.1).OR.(FS_STATE.EQ.2))  FBTOSB(,) = FBTOSB(,+)
      FS_STATE = FS_STATE + 1
      IF (FS_STATE .GT. 2)   FS_STATE = 0
      FS_ACTIVE = (FS_STATE .EQ. 0)
C
```

```
C   This is the added bit for the implementation with counters.
C
C   We first transfer the control signals that will tell the processors
C   their counters have reached their initial values.
C
      INIT_C_STOP = INIT_C_STOP(,+)
C
C   Notice that as with all the other control signals, we make the
C   transfer in the horizontal (,+) direction.  We could just as well
C   make it in the vertical direction (+,), it would not make any
C   difference.
C
C   Now we make sure the processors currently holding the I...STOP
C   control signal notice its presence...  The .FALSE. in this next
C   line means that the processors must not increment their counter
C   anymore to initialise them (because they are supposed to be
C   initialised by now).
C
      INIT_COUNT (INIT_C_STOP) = .FALSE.
C
C   When a processor has received the Fast to Slow belt transfer
C   signal, it should start modifying its counters.  Here, we have
C   the processor note this.  The processors at an even distance
C   from the base (they never do a transfer from Fast to Slow... )
C   notice the presence of the control signal while this one is
C   inactive.
C
      IF ((FS_STATE.EQ.0).OR.(FS_STATE.EQ.2))
     X                UPDATE_COUNT(FBTOSB) = .TRUE.
C
C   When a processor puts the value in its accumulator on (in) the fast
C   belt (register), that's its value computed.  So after that happens,
C   there is no point (neither harm) in continuing to modify the
C   counters.  The ACCumulator TO Fast Belt control signal indicates
C   when this happen and here we use this signal to reset the
C   UPDATE_COUNT bit.
C
      IF (AF_STATE.EQ.0)   UPDATE_COUNT(ACCTOFB) = .FALSE.
      RETURN
C
C   END SUBROUTINE   * *  U P D A T E _ S I G N A L S  * *
C
      END
```

```
C          ****************************************************
C          *                                                  *
C          *   SUBROUTINE   M A R K _ P R O C E S S O R S     *
C          *                                                  *
C          ****************************************************

      ENTRY SUBROUTINE  MARK_PROCESSORS
C
C   This subroutine implement the marking phase of the "array"
C   parsing algorithm.  During this phase, pointers obtained by
C   the previous phase are used to mark the nodes of the
C   underlying parse tree as "tree" nodes and those in between
C   as "link" nodes.  This is done by token passing.  The root
C   initiates the marking.  It sends on eah beat two tokens, one
C   vertically and one horizontally.  It sends in one direction
C   as many tokens as the value of the pointer for this
C   direction.  If a token gets two tokens in a row, it marks
C   itself as a "link" node.  If it gets one first token and
C   then none on the following beat, it marks itself as a "tree"
C   node.  Once a tree node got marked, it initiates the marking
C   of its own subtree.
C
C   We also record in which direction came the tokens.
C
C   This is the phase 3 (PH3DAP) version of MARK_PROCESSORS.  In
C   this version, we do two things.  First, we color the
C   processors with black and white so as to obtain a chess
C   board pattern.  The goal is to identify the processors on
C   even and odd levels.  Second, because our grammar is
C   "locally ambiguous", we need to pass down with the tokens,
C   the non-terminals of the right-hand side of the rule of the
C   processor.  These non-terminals will tell the sons which
C   rule to choose (the one with the non-terminal passed down as
C   the left-hand side) among rules with a common right-hand
C   side.  (Not all sons will have used such rule to compute
C   "their value".)  Before the marking takes place, we assign
C   to the relevant variables the values of the two non-terminal
C   of the right-hand sides used during recognition. After all
C   the marking is over, we determine the correct rules
C   associated with each tree node processor.
C
      LOGICAL  TEMP_TOKEN(,),    V_TOKEN_1(,),     V_TOKEN_2(,),
     X                           H_TOKEN_1(,),     H_TOKEN_2(,),
     X         VERTICAL(,),      LINK_NODE(,),     TREE_NODE(,)
      CHARACTER   CHAR_MARK(,)
      EQUIVALENCE (TEMP_TOKEN,CHAR_MARK)
```

```
      COMMON  /MARKING/   TEMP_TOKEN, V_TOKEN_1, V_TOKEN_2,
     X                                H_TOKEN_1, H_TOKEN_2,
     X                    VERTICAL,   LINK_NODE, TREE_NODE

C TEMP_TOKEN  : temporary bit for token transfers (passings).
C V_TOKEN_1   : vertical token received on "this" beat.
C V_TOKEN_2   : horizontal token received on the "previous" beat.
C H_TOK...
C VERTICAL    : indicates if the token that were received (if any)
C               were coming in the vertical direction.
C LINK_NODE   : the actual "link" node mark.
C TREE_N...
C CHAR_MARK   : a character matrix EQUIVALENCEd with the stuff above
C               for the purpose of transfering the information back
C               to the host.

      LOGICAL      BEAT_IS_ODD
      INTEGER      I

C BEAT_IS_ODD : we test if processors can be marked on every other
C               beat.  This is what tells us if we are on even beats
C               or not.
C I           : (useless) loop control variable.

      INTEGER     NTOKEN,  RESULT
      INTEGER*2  ACCSIG(,)
      COMMON /ARGMTS/   NTOKEN, RESULT, ACCSIG
      LOGICAL     INIT_COUNT(,),  UPDATE_COUNT(,)
      INTEGER*2  BIG_COUNT(,),   WEE_COUNT(,),
     X           V_POINTER(,),   H_POINTER(,)
      COMMON /COUNTING/  INIT_COUNT, UPDATE_COUNT,
     X                   BIG_COUNT,  WEE_COUNT,
     X                   V_POINTER,  H_POINTER

C   See subroutine RECOGNIZE for a description of these variables.
C
C   Added for phase 3.
C
      INTEGER*1     LEFT_SIDE(,),  RIGHT_SIDE_1(,),  RIGHT_SIDE_2(,),
     X              RULE(,)
      LOGICAL       ODD_LEVEL(,)
      COMMON  /RULING/  LEFT_SIDE, RIGHT_SIDE_1, RIGHT_SIDE_2,
     X                  RULE,      ODD_LEVEL

C   See subroutine ACC_UPDATE for a description of these variables.
C
```

```
C   Start of the body of subroutine  MARK_PROCESSORS  .
C
C
C   We don't need the following since plane geometry is on by
C   default but just in case.
C
      GEOMETRY(PLANE,PLANE)
C
C   We initialise everything to false except the tree node mark at the
C   root (the root of the tree is processor (1,1)).
C
      TEMP_TOKEN    =  .FALSE.
      V_TOKEN_1     =  .FALSE.
      V_TOKEN_2     =  .FALSE.
      H_TOKEN_1     =  .FALSE.
      H_TOKEN_2     =  .FALSE.
      VERTICAL      =  .FALSE.
      LINK_NODE     =  .FALSE.
      TREE_NODE     =  .FALSE.
      TREE_NODE(1,1)=  .TRUE.
C
C   Before we start the marking (this is the phase 3 version), we set
C   the variables RIGHT_SIDE_1 and RIGHT_SIDE_2 according to the rules
C   recorded by the processors during recognition.
C
C   We repeat here the convention for non-terminal numbering:
C
C   non-terminals:    E    T    F   E)   +T   *F   <(>  <)>  <+>  <*>
C         numbers:    1    2    3    4    5    6    7    8    9   10
C
C rule (1)    E  -->  E  +T

      RIGHT_SIDE_1(RULE.EQ.1) =  1
      RIGHT_SIDE_2(RULE.EQ.1) =  5

C rule (2)    E  -->  T  *F

      RIGHT_SIDE_1(RULE.EQ.2) =  2
      RIGHT_SIDE_2(RULE.EQ.2) =  6

C rule (3)    E  --> <(> E)

      RIGHT_SIDE_1(RULE.EQ.3) =  7
      RIGHT_SIDE_2(RULE.EQ.3) =  4

C rule (5)   E)  -->  E  <)>
```

```
      RIGHT_SIDE_1(RULE.EQ.5) =  1
      RIGHT_SIDE_2(RULE.EQ.5) =  8

C rule (9)   +T  --> <+>  T

      RIGHT_SIDE_1(RULE.EQ.9) =  9
      RIGHT_SIDE_2(RULE.EQ.9) =  2

C rule (12)  *F  --> <*>  F

      RIGHT_SIDE_1(RULE.EQ.12) =  10
      RIGHT_SIDE_2(RULE.EQ.12) =  3
C
C   We are ready for our one and only loop.  We start with beat 1 which
C   is odd, right?
C
      BEAT_IS_ODD   =  .TRUE.
C
C   We circle around two times NTOKEN minus 2.  The thing is the
C   furthest processor from the root is (NTOKEN-1) and it takes
C   2n beat to mark a processor at a distance n from the root.
C   Because DAP FORTRAN does not like loop control variables to
C   be less than zero we have to had a stupid test just before
C   the loop.
C
      IF (NTOKEN.EQ.1)  GOTO 101
      DO  100  I = 1 , 2 * (NTOKEN-1)
C
C   It's quite simple, first pass the tokens around (twice) then
C   test for marking.  We start in the vertical direction.
C
C
C   Well, passing tokens around is not quite so simple.  Here's
C   the situation: tree nodes pass down a token whenever their
C   associated pointer is strictly positive; link node pass down
C   a token if they have received a token during the two
C   previous beats (which is the same as saying that they keep
C   the first token they receive and pass down the other ones on
C   the beat after they receive them.).  To implement this, we
C   have a two stage pipeline.  Token come in one end of the
C   pipeline and they come out the other end only if the
C   pipeline is full.  The variables ..._TOKEN_1 and ..._TOKEN_2
C   implement the pipeline.  The pipeline can be full only for
C   link nodes, so when we test for a full pipeline, we don't
C   need to test only on link nodes.  You got all that?  O.K.,
```

```
C   here's what we do: We find out if the processors can pass
C   down a token; we shift the content of the first stage of the
C   pipeline into the second stage; we then pass the tokens from
C   one processor to the next (in the first stage of the
C   pipeline).
C
C
C   Find out if the processors have a token to pass down.  And while
C   you're at it, decrement the relevant pointer in the tree nodes.
C
      TEMP_TOKEN =    (V_TOKEN_1.AND.V_TOKEN_2)
     X              .OR.(TREE_NODE.AND.(V_POINTER.GT.0))
      V_POINTER(TREE_NODE .AND. (V_POINTER.GT.0)) = V_POINTER-1
C
C   Shift the first stage of the pipe.
C
      V_TOKEN_2  =  V_TOKEN_1
C
C   Pass down (transfer) the tokens (and non-tokens).
C
      V_TOKEN_1  =  TEMP_TOKEN(-,)
C
C   We set the VERTICAL bit of any processor that has received a token
C   just now (although VERTICAL is only useful for link nodes).
C
      VERTICAL =  VERTICAL .OR. V_TOKEN_1
C
C   Same thing, but horizontally now.  Of course, this time, we
C   do not set the VERTICAL logical bit.
C
C
C   Find out if the processors have a token to pass down.  And while
C   you're at it, decrement the relevant pointer in the tree nodes.
C
      TEMP_TOKEN =    (H_TOKEN_1.AND.H_TOKEN_2)
     X              .OR.(TREE_NODE.AND.(H_POINTER.GT.0))
      H_POINTER(TREE_NODE .AND. (H_POINTER.GT.0)) = H_POINTER-1
C
C   Shift the first stage of the pipe.
C
      H_TOKEN_2  =  H_TOKEN_1
C
C   Pass down (transfer) the tokens (and non-tokens).
C
      H_TOKEN_1  =  TEMP_TOKEN(,-)
C
```

```
C   If we have passed tokens around twice (if we are on a even
C   beat), we test to see if we have not identified link nodes
C   and tree nodes.  We have identified a link node if a
C   processor has received a first token and then a second one.
C   We have identified a tree node if a processor received a
C   first token but not a second one.  Of course, we mark the
C   processors we identify as tree node or link node (that the
C   whole point of this subroutine).
C
C
C   First of all, test if we are on an even beat.  Otherwise, we won't
C   bother with all this.
C
      IF (BEAT_IS_ODD)  GOTO 99
C
C   A processor is identified as a link node if either of its token
C   pipeline is full.  That then means that the processor has received
C   tokens on two successive beats.  Of course, a processor that has
C   already been marked as a link node remains marked that way.
C
      LINK_NODE =  LINK_NODE .OR. (V_TOKEN_1 .AND. V_TOKEN_2)
     X                       .OR. (H_TOKEN_1 .AND. H_TOKEN_2)
C
C   A processor is identified as a tree node if only the second
C   stage of its pipeline contains a token.  That then means
C   that the processor received a token on the second previous
C   beat but none on the previous.  Of course, a processor that
C   has already been marked as a tree node remains marked that
C   way whatever and also, a processor that has been marked as a
C   link node does not subsequently get marked as a tree node.
C
      TREE_NODE =  TREE_NODE .OR.
     X             (.NOT. LINK_NODE .AND.
     X                  (     (.NOT.V_TOKEN_1 .AND. V_TOKEN_2)
     X                   .OR.(.NOT.H_TOKEN_1 .AND. H_TOKEN_2)))
C
C   Next bit added for phase 3.  We need to pass down with the
C   tokens information originating from the father and destined
C   to its son.  This information is the non-terminal sent by
C   the son (among others) during recognition that the father
C   actually used to compute its "value" (set of non-terminals).
C   This information has been extracted from the rule the father
C   recorded (see the code before this loop).  Here, we pass
C   DOWN the FIRST non-terminal of this right-hand side.  The
C   non-terminal is only sent to link nodes.  The idea is to
C   make sure that at the end of the marking phase, the sons can
```

```
C   pick-up from their immediate neighbor linking them to their
C   father this non-terminal.
C
      RIGHT_SIDE_1(LINK_NODE.AND.VERTICAL)      =  RIGHT_SIDE_1(-,)
C
C   We pass to the RIGHT the SECOND non-terminal of the nearest
C   tree node on the left.
C
      RIGHT_SIDE_2(LINK_NODE.AND..NOT.VERTICAL) =  RIGHT_SIDE_2(,-)
C
C   We color the processors according to their level parity (odd or
C   even).  For this, we color a processor according to the opposite
C   parity of its neighbor to its left or the one above (which should
C   have the same parity if they both exist).
C
C   Observe that we did no initialisation before the loop for this
C   coloring.  We dont need any because we relie on the PLANE GEOMETRY
C   which feeds in .FALSE. logical values on the boundary.  Before the
C   loop, all the level parity are undefined.  Each (even) loop cycle
C   sets properly the level parity of one level.  (We'll need to do
C   what follows one more time outside the loop.)  This simple line
C   of code (imbedded in a loop) is a bit tricky.  The reader might
C   need a bit of thinking about it to see what's going on...
C
      ODD_LEVEL =  .NOT.(ODD_LEVEL(-,).OR.ODD_LEVEL(,-))
C
C   That's almost the end of this loop.  Each turn of a loop is a
C   beat so before we finish this turn, we toggle the beat parity.
C
   99 BEAT_IS_ODD =  .NOT. BEAT_IS_ODD

  100 CONTINUE

  101 CONTINUE

C   Old stuff below.  This is the phase 3 version so we'll put the
C   following in comments and add something afterwards.
C
C
C   This loop constitute just about the whole of this
C   subroutine.  We simply need before returning to convert the
C   data we've been working on from DAP format to HOST FORTRAN
C   format.  We use a CHARACTER matrix EQUIVALENCEd to all
C   LOGICAL matrix we've used.  Because we are going to work on
C   this "CHARACTER" matrix in the Host part of the program, (I
C   am not sure but I think) we need to make sure the most
```

```
C   significant bit of all the characters are set to zero
C   (otherwise we would have undefined character codes).  We do
C   so by reseting TEMP_TOKEN.
C
C
C      TEMP_TOKEN =  .FALSE.
C
C      CALL  CONVMF1 (CHAR_MARK)
C
C   Next bits added for phase 3.  We first set the level parity of
C   the last level, level NTOKEN.  (See comment inside the loop.)
C
      ODD_LEVEL =  .NOT.(ODD_LEVEL(-,).OR.ODD_LEVEL(,-))
C
C   The sons that used right-hand sides common to more than one
C   rule determine here which rule was relevant.
C   First, they get the non-terminal number of their left-hand side.
C
      LEFT_SIDE(TREE_NODE.AND.     VERTICAL) = RIGHT_SIDE_1(-,)
      LEFT_SIDE(TREE_NODE.AND..NOT.VERTICAL) = RIGHT_SIDE_2(,-)
C
C   Now, from their left-hand sides, the processors with incorrect rule
C   recorded determine their correct rule.  Remember:
C
C     non-terminals:     T    F
C          numbers:      2    3
C
C  rule (2)  E  -->  E  +T,  rule (6)  T --> ...
C
C  rule (2) is recorded during recognition and we switch to rule (6)
C  only if the left-side is "T".
C
      RULE((RULE.EQ.2).AND.(LEFT_SIDE.EQ.2)) =  6

C  rule (3)  E  --> <(> E), rule (7) T --> ..., rule (10)  F --> ...

      RULE((RULE.EQ.3).AND.(LEFT_SIDE.EQ.2)) =  7
      RULE((RULE.EQ.3).AND.(LEFT_SIDE.EQ.3)) =  10

C  rule (4)  E  -->  a,  rule (8)  T -->  a,  rule (11)  F -->  a

      RULE((RULE.EQ.4).AND.(LEFT_SIDE.EQ.2)) =  8
      RULE((RULE.EQ.4).AND.(LEFT_SIDE.EQ.3)) =  11
      RETURN
C
C   END of subroutine   * * *   M A R K _ P R O C E S S O R S   * * *
```

```
C
      END


C            ******************************************************
C            *                                                    *
C            *    SUBROUTINE      O U T P U T _ P A R S E         *
C            *                                                    *
C            ******************************************************

      SUBROUTINE  OUTPUT_PARSE
C
C       This subroutine implement the third phase of the algorithm.  In
C   this phase, the parse present in the array reconfigured as a tree
C   is output through the root of the tree, processor (1,1).  This is
C   achieved by a kind of structured bucket brigade.  Every processors
C   try to pass rules up the tree.  Processors however never pass a
C   rule up if the processor they want to pass it to is not ready.  To
C   control the whole thing, all we need is to make all the processors
C   go through the following automata:
C
C                       .------. 0 0
C                       |      |
C         -----------------    |
C        |                 |   |
C        |  State A -rule- | <--
C  0 1   |                 |        0 1               1 1
C    --> |       0 0       | <-------.         .------.
C   |    |                 |         |         |      |
C   |     -----------------      -----------------    |
C   |                  |        |                 |   |
C   |            1 0  -------> |  State B -rule- | <--
C   |                           |                 |
C   |            1 1  -------> |      1 1        |
C   |                  |        |                 |
C   |     -----------------      -----------------
C   |    |                 |         |
C    --- |  State C   - -  | <--------
C        |                 |          0 0 , 1 0
C        |       1 0       | <--.
C        |                 |    |
C         -----------------     |
C                       |       |
C                        -------
C                               0 0 , 1 0
C
```

```
C   Let me explain.  On every beat, a set of processors send things to
C   neighbors and of course, a set of processors receive things.  On
C   the next beat, the roles are inversed.  Processors on sending beats
C   send two things, rules (up) and requests (down).  A processor can
C   send a rule up only if it received a request on the previous beat.
C   In the diagram you see above, there are labels composed of two bits
C   (one label has two pair of bits).  On transition labels, the first
C   bit of the pair indicates if on last receiving beat the processor
C   received a request and the second indicates if it received a rule.
C   In a state rectangle label, the pair indicates what the processor
C   will send on the next sending beat (using the same convention).
C   Processors on even levels are on sending beat when those on odd
C   levels are on receiving beats (and vice versa).  (See subroutine
C   ACC_UPDATE comments for a description of levels.)
C
C   Before we go on, we have to explain one last thing.  During the
C   output, link nodes just pass up whatever they receive.  Tree nodes
C   are a bit more sophisticated.  They first pass up the rule they
C   have recorded.  Then, they pass the rules coming from their left
C   branch (verticaly) until one comes with a "last-rule" marker.  They
C   strip this marker before passing up the rule and then, they pass
C   the rules  coming from their right branch.  When they receive a
C   rule marked as the last from this branch, they pass up this rule
C   with the marker this time and then "stop".
C
C   Now let's implement all this nice stuff.
C
      INTEGER    NTOKEN, RESULT
      INTEGER*2  ACCSIG(,)

C   If DAP FORTRAN had the PARAMETER statement:
C
C      LOGICAL    SIGACC(,,16),
C                 INIT_C_STOP(,), ACCTOFB(,), FBTOSB(,),
C                 ACCUM(,,NOFNTERM)
C
C   But it does not so:

      LOGICAL    SIGACC(,,16),
     X           INIT_C_STOP(,), ACCTOFB(,), FBTOSB(,),
     X           ACCUM(,,10)
      LOGICAL    BASE_NODE(,)
      EQUIVALENCE (SIGACC,ACCSIG),
     X            (SIGACC(,,3),BASE_NODE),
     X            (SIGACC(,,4),INIT_C_STOP),
     X            (SIGACC(,,5),ACCTOFB), (SIGACC(,,6),FBTOSB),
```

```
     X           (SIGACC(,,7),ACCUM)
      COMMON /ARGMTS/   NTOKEN, RESULT, ACCSIG

C See subroutine RECOGNIZE for a description of these variables.
C       (Only BASE_NODE is of interest to us in this subroutine.)

      INTEGER*1    LEFT_SIDE(,),  RIGHT_SIDE_1(,),  RIGHT_SIDE_2(,),
     X             RULE(,)
      LOGICAL      ODD_LEVEL(,)
      COMMON  /RULING/  LEFT_SIDE, RIGHT_SIDE_1, RIGHT_SIDE_2,
     X                  RULE,      ODD_LEVEL

C   See subroutine ACC_UPDATE for a description of these variables.

      LOGICAL  TEMP_TOKEN(,),     V_TOKEN_1(,),     V_TOKEN_2(,),
     X                            H_TOKEN_1(,),     H_TOKEN_2(,),
     X         VERTICAL(,),       LINK_NODE(,),     TREE_NODE(,)
      CHARACTER   CHAR_MARK(,)
      EQUIVALENCE (TEMP_TOKEN,CHAR_MARK)
      COMMON  /MARKING/   TEMP_TOKEN, V_TOKEN_1, V_TOKEN_2,
     X                                H_TOKEN_1, H_TOKEN_2,
     X                    VERTICAL,   LINK_NODE, TREE_NODE

C See subroutine MARK_PROCESSORS for a description of these variables.

      INTEGER*1    STATE(,),      NEWSTATE(,),    RULE_BUFFER(,)
      INTEGER      RULE_INDEX
      LOGICAL      RULE_OUT(,),  REQUEST_OUT(,), LAST_RULE(,),
     X             RULE_IN (,),  REQUEST_IN (,), RECEIVING(,),
     X             BEAT_IS_EVEN, FINISHED

C STATE        : the states of the processors.
C NEWSTATE     : a temporary for the computation of next states.
C RULE_BUFFER  : buffer for the passing of the rules by the bucket
C                brigade.
C RULE_INDEX   : just a vector index that will tell us where to store
C                the next rule of the parse.
C RULE_OUT     : indicates (to the interested neighbor) that this
C                processor will send a rule on this sending beat.
C RULE_IN      : indicates that the interested neighbor will receive
C                the rule just mentionned.
C REQUEST_ ...
C LAST_RULE    : indicates that this rule is the last one to come from
C                this branch.
C RECEIVING    : indicates that the processor is on a receiving beat.
C BEAT_IS_EVEN : the usual...
```

```
C FINISHED     : tells us when we are finished with the output of the
C         parse.  Not necessary but makes the program more readable.

      INTEGER*1    PARSE(127)
      INTEGER      OUTPUT_BEATS
      COMMON  /PARSING/ PARSE, OUTPUT_BEATS

C PARSE        : a vector that will receive the parse, the list of
C                rules used to derive the input string from the
C                starting symbol (or whatever).
C OUTPUT_BEATS : depending on the shape of the parse tree, we will need
C                a different number of beats to output a parse.  With
C                this variable, we will count how many we needed.
C PARSING      : common block to pass the parse to the Host.


C
C  Start of the body of subroutine  * *  O U T P U T _ P A R S E  * *
C
C
C   Of course, we start with initialisations.
C
C
C   State initialisations.  The link nodes are in state C (3), no rule
C   to send and no request received.  The tree nodes are in state A(1),
C   their own rule to send (because we output in leftmost order) but no
C   request received.  The root is in state B (2), its rule to send and
C   it's just as though we had sent it a request for it.  All the other
C   nodes are in no state (0).
C
      STATE =  0
      STATE(LINK_NODE) =  3
      STATE(TREE_NODE) =  1
      STATE(1,1)       =  2
C
C   Processors on even levels are going to be
C   a on receiving beat first.
C
      RECEIVING =  .NOT. ODD_LEVEL
C
C   The tree nodes send their own rule first so they put it now
C   in their rule buffer.
C
      RULE_BUFFER(TREE_NODE) =  RULE
C
C   The root has a rule in its buffer.  We can pick it up at the start
```

```
C   of the output loop.  To be informed of this fact we set its RULE_IN
C   logical bit.
C
      RULE_IN(1,1) =  .TRUE.
C
C   The base processors are leaf (tree) nodes.  They have no subtree so
C   their own rule is the last one "passing through" them.
C
      LAST_RULE =  BASE_NODE
C
C   The VERTICAL variable has a slightly different meaning then
C   in the MARK_PROCESSORS subroutine.  In that sub., it meant
C   that a processor had received a token from "above" either
C   VERTICALy or horizontally.  Here, it means that a processor
C   receives rules from "below" VER... .  The VERTICAL logical
C   bits were set in MARK_PROCESSORS.  This setting is ok for
C   the link nodes.  For tree nodes, they all start in VERTICAL
C   mode (because we output in leftmost order).  They change
C   when they have output their whole left sub-parse.  (Link
C   node never Change mode.
C
      VERTICAL(TREE_NODE) =  .TRUE.
C
C   Up to now, we have put no rule in the PARSE vector, have we?
C   And we have executed no beat either.
C
      RULE_INDEX   =  0
      OUTPUT_BEATS =  0
C
C   We are ready now to start the loop.  Beat one isn't even, is it?
C
      BEAT_IS_EVEN = .FALSE.
C
C   The loop is of the sort in between a "while" and a "repeat" loop,
C   i.e. the test is in the  middle of the body (and it is executed
C   only on odd beats).
C
  100 OUTPUT_BEATS =  OUTPUT_BEATS + 1
C
C   On odd beats, we pick up the rule in the root rule buffer if there
C   is any in there.
C
      IF (BEAT_IS_EVEN .OR. .NOT.RULE_IN(1,1))  GOTO  200
      RULE_INDEX        =  RULE_INDEX + 1
      PARSE(RULE_INDEX) =  RULE_BUFFER(1,1)
C
```

```
C   Here's the test for exit of the loop.
C
      IF (LAST_RULE(1,1))  GOTO 300

  200 CONTINUE
C
C   We first see which processors are sending rules and requests.
C
      RULE_OUT    =  (STATE.EQ.2) .AND. (.NOT.RECEIVING)
      REQUEST_OUT =  ((STATE.EQ.2) .OR. (STATE.EQ.3))
     X                            .AND. (.NOT.RECEIVING)
C
C   Now, the complement code, which processors receive these rules and
C   requests.  Note that as a special case, we send a request to the
C   root.  (This request will be used only for the "next state"
C   computation and only on beats when the root is receiving.)
C
      RULE_IN    = (      VERTICAL .AND. RULE_OUT(+,)) .OR.
     X             (.NOT.VERTICAL .AND. RULE_OUT(,+))
      REQUEST_IN = (      VERTICAL(-,) .AND. REQUEST_OUT(-,)) .OR.
     X             (.NOT.VERTICAL(,-) .AND. REQUEST_OUT(,-))
      REQUEST_IN(1,1) =  .TRUE.
C
C   Now we can proceed to the exchange of rules as such.
C
      RULE_BUFFER(RULE_IN .AND.      VERTICAL) =  RULE_BUFFER(+,)
      RULE_BUFFER(RULE_IN .AND. .NOT.VERTICAL) =  RULE_BUFFER(,+)
C
C   These last few statements were pretty straightforward.  The next
C   bit is kind of tricky.  Here's what's happening.  Link node just
C   transmit rules and "last rule" markers.  Horizontal mode tree node
C   do the same but VERTICAL mode tree node dont transmit a last rule
C   marker.  That's not all,  if asked to do so, they go in horizontal
C   mode.  Recall, tree nodes transmit first vertically until they get
C   a "last rule" marker which they leave hanging.  Then they
C   transmit horizontally until they get the marker again,  but this
C   time they transmit the marker.
C
C
C   Horizontal nodes, either link or tree, transmit any "last rule"
C   marker received.
C
      LAST_RULE(.NOT.VERTICAL .AND. RULE_IN)=  LAST_RULE(,+)
C
C   Vertical link nodes transmit any ...
C
```

```
      LAST_RULE(LINK_NODE .AND. VERTICAL .AND. RULE_IN)=  LAST_RULE(+,)
C
C   Vertical tree nodes presented with a "last rule" marker go in the
C   horizontal mode (and ignore the marker).
C
      VERTICAL(TREE_NODE .AND. VERTICAL .AND.
     X         RULE_IN   .AND. LAST_RULE(+,)) =  .FALSE.
C
C   We now execute the state transitions.  Processors change state
C   according to what they have received from their neighbors, so only
C   processors RECEIVING change states.  We set the variable NEW_STATE
C   of every processors but this variable will change the values of the
C   variable STATE of only the RECEIVING processors.
C
C   We do not explicitly implement every transitions of the FSM
C   depicted in the comment at the beginning of this subroutine.
C   Transitions to the same state are implemented by the fact that we
C   assign to NEW_STATE the value of STATE to start with.
C
C   To make the following code a bit more clear, let me describe
C   informally what each state "means":
C
C      (1)  A  -  The processor has a rule to send up but has not
C                 received a request for it.  It will wait for a
C                 request.
C      (2)  B  -  The processor has a rule to send and received a
C                 request for it.  It will send the rule and send
C                 a request for another one.
C      (3)  C  -  The processor has no rule to send up.  It will send
C                 requests for one until it gets one.
C
      NEW_STATE = STATE
      NEW_STATE( (STATE.EQ.1)  .AND.      REQUEST_IN) =  2
      NEW_STATE( (STATE.EQ.2)  .AND. .NOT.RULE_IN)    =  3
      NEW_STATE( (STATE.EQ.3)  .AND.      RULE_IN
     X                         .AND.      REQUEST_IN) =  2
      NEW_STATE(((STATE.EQ.2).OR.
     X           (STATE.EQ.3)) .AND.      RULE_IN
     X                         .AND. .NOT.REQUEST_IN) =  1
      STATE(RECEIVING) = NEW_STATE
C
C   We come to the end of our loop.  We'll start another cycle but
C   before we do so, we will toggle the RECEIVING mode of the
C   processors.  Those that were RECEIVING will become "sending"
C   and ...  Oh, and we toggle the beat parity (of course).
C
```

```
      RECEIVING    =  .NOT. RECEIVING
      BEAT_IS_EVEN =  .NOT. BEAT_IS_EVEN
      GOTO  100

  300 CONTINUE
C
C   That's the parse in the vector PARSE now.  The DAP part of the
C   program is just about finished.  All that is left to do is to
C   convert this vector(*) to Host FORTRAN format so as to allow the
C   Host part to present the results.  We also want to present
C   OUTPUT_BEATS to the user but this requires no conversion.
C
C    *** PARSE is a one dimensional array of "scalar" elements
C        and so, following DAP FORTRAN terminology, the mode of
C        PARSE is "Scalar", not "Vector".  Hence, we use the
C        conversion subroutine "convSf1", not "convVf1".
C
      CALL CONVSF1 (PARSE,127)
      RETURN
C
C   END of subroutine    * *  O U T P U T _ P A R S E  * *
C
      END



C           ****************************************************
C           *                                                  *
C           *     ENTRY SUBROUTINE      D A P _ P H A S E S    *
C           *                                                  *
C           ****************************************************

      ENTRY SUBROUTINE  DAP_PHASES
C
C   This, would it not be for INIT_ARRAY, would be the only
C   subroutine called by the Host.  What it actually does is it
C   calls the DAP routine for the Host.  We do things that way
C   because we want to minimise the overhead involved in calling
C   the DAP from the Host.
C
C   Added bit for the Phase 3 version.  (It should have been there
C   for Phase 2!).  We test if recognition is succesful and only if it
C   is, do we call the other phases.  An unsuccessfull will be
C   indicated by a negative value in OUTPUT_BEATS.
C
      INTEGER*1    LEFT_SIDE(,),  RIGHT_SIDE_1(,),  RIGHT_SIDE_2(,),
     X             RULE(,)
```

```
      LOGICAL       ODD_LEVEL(,)
      COMMON  /RULING/  LEFT_SIDE, RIGHT_SIDE_1, RIGHT_SIDE_2,
     X                  RULE,      ODD_LEVEL

C See subroutine ACC_UPDATE for a description of these variables.

      INTEGER*1     PARSE(127)
      INTEGER       OUTPUT_BEATS
      COMMON  /PARSING/ PARSE, OUTPUT_BEATS

C See subroutine OUTPUT_PARSE for a description of these variables.
C
C   Start of the body of subroutine   * *  D A P _ P H A S E S  * *
C
      CALL  RECOGNIZE
      OUTPUT_BEATS = -1
C
C   If the root, processor (1,1) has not recorded a rule during
C   recognition, nothing has been recognized and we do not call the
C   other two phases.  OUTPUT_BEATS at value -1 will indicate this.
C
      IF (RULE(1,1).EQ.0)  RETURN
      CALL  MARK_PROCESSORS
      CALL  OUTPUT_PARSE
      RETURN
C
C   END of subroutine    * *   D A P _ P H A S E S   * *
C
      END
```

Bibliography

[Aho 72] A. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling: Parsing*, volume 1. Prentice-Hall, Englewood Cliffs, N. J., 1972.

[Aho 74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Computer Science and Information Processing. Addison-Wesley, Reading, Mass., 1974.

[Arlazarov 70] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR*, 194:487–488, 1970. (In Russian) English translation in *Soviet Math. Dokl.* **11**:5, 1209–1210.

[Austin 79] J. H. Austin Jr. The burroughs scientific processor. In C. R. Jesshope and R. W. Hockney, editors, *Infotech State of the Art Report: Supercomputers*, pages 1–31. Infotech International Ltd., 1979.

[Babbage 22] Charles Babbage. Respecting the application of machinery to the calculation of astronomical tables. *Memoirs of the Astronomical Society*, 1:309, 1822.

[Beyer 69] W. T. Beyer. Recognition of topological invariants by iterative arrays. Project MAC Technical Report TR-66, MIT, Cambridge, 1969.

[Birtwistle 73] G. M. Birtwistle, O. J. Dahl, B. Myhraug, and K. Nygaard. *SIMULA begin;*. Auerbach, Philadelphia, 1973.

[Burks 70] In Arthur W. Burks, editor, *Essays on Cellular Automata.* University of Illinois Press, Urbana, Illinois, 1970.

[Chang 87] J. H. Chang, O. H. Ibarra, and M. A. Palis. Parallel parsing on a one-way array of finite-state machines. *IEEE Transactions on Computers*, 36(1):64–75, January 1987.

[Chiang 84] Y. T. Chiang and K. S. Fu. Parallel parsing algorithms and VLSI implementations for syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(3):302–314, May 1984.

[Chomsky 59] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):127–167, 1959.

[Clocksin 81] W. F. Clocksin and C. S. Mellish. *Programming in PROLOG.* Springer-Verlag, Berlin Heidelberg New York, 1981.

[Codd 68] E. F. Codd. *Cellular Automata.* ACM Monograph Series. Academic Press, New York, 1968.

[Cole 69] S. N. Cole. Real time computation by n-dimensional iterative arrays of finite-state machines. *IEEE Transactions on Computers*, 18:349–365, 1969.

[Cole 84] M. I. Cole. Partitioning programmable logic arrays. Internal Report CSR-166-84, University of Edinburgh, Department of Computer Science, October 1984.

[De Michelli 83] G. De Michelli and A. L. Sangiovanni-Vincentelli. PLEASURE: a computer program for simple and multiple constrained folding of programmable logic arrays. In *20th Design Automation Conference Proceedings*, pages 530–537, Miami Beach, Florida, June 1983. ACM/IEEE.

[Deker 86] E. Deker, S. Peng, and S. S. Iyengar. Optimal parallel algorithms for construction and maintaining a balanced m-way search tree. *International Journal of Parallel Programming*, 15(6):503–528, Dec 1986.

[Earley 68] J. Earley. *An Efficient Context-Free Parsing Algorithm.* PhD thesis, Carnegie-Mellon University, 1968.

[Earley 70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[Fischer 75] C. N. Fischer. *On Parsing Context Free Languages in Parallel Environments.* PhD thesis, Cornell University, 1975.

[Fischer 80] C. N. Fischer. On parsing and compiling arithmetic expressions on vector computers. *ACM Transactions on Programming Languages and Systems*, 2(2):203–224, Apr 1980.

[Floyd 61] R. W. Floyd. A descriptive language for symbol manipulation. *Journal of the ACM*, 8(4):579–584, 1961.

[Frank 79] P. D. Frank. *Bounded Nondeterminism and the Parallel Parsing of Context-Free Languages.* PhD thesis, University of Washington, 1979.

[Frenkel 86] Karen A. Frenkel. Evaluating two massively parallel machines. *Communications of the ACM*, 29(8):752–758, Aug 1986.

[Goldberg 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Goldstine 46] H. H. Goldstine and A. Goldstine. The electronic numerical integrator and computer (ENIAC). *Mathematical Tables and Aids to Computation*, 2(15):97–110, 1946.

[Gostick 79] R. W. Gostick. Software and algorithms for the distributed-array processors. *ICL Technical Journal*, pages 114–135, May 1979.

[Graham 76a] S. L. Graham and M. A. Harrison. Parsing of general context-free languages. *Advances in Computers*, 14, 1976.

[Graham 76b] S. L. Graham, M. A. Harrison, and W. L. Ruzzo. On line context free language recognition in less than cubic time. In *Proceedings of the 8th Annual ACM Symposium on the Theory of Computer Science*, pages 77–185. ACM, May 1976.

[Guibas 79] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Caltech Conference on VLSI*, pages 509–525, Jan 1979.

[Hamacher 68] V. C. Hamacher. *A class of parallel processing automata.* PhD thesis, Syracuse University, 1968.

[Harper 86] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Internal Report ECS-LFCS-82-2 (CSR-209-86), University of Edinburgh, Department of Computer Science, 1986.

[Hartree 46] D. R. Hartree. The ENIAC, an electronic computing machine. *Nature*, 158:500–506, 1946.

[Hays 67] D. G. Hays. *Introduction to Computational Linguistics.* American Elsevier, New York, 1967. In this book, Hays describes the CYK algorithm and attributes it to J. Cocke.

[Heller 78] D. Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM*, 20:740–777, 1978.

[Higbie 79] L. C. Higbie. Vectorisation and conversion of FORTRAN programs for the CRAY-1 (CFT) compiler. Document 2240207, Cray Research Inc., 1979.

[Hillis 85] Daniel W. Hillis. *The Connection Machine.* MIT Press, Cambridge, Massachussetts, 1985.

[Hirakawa 83] H. Hirakawa. Chart parsing in concurrent prolog. Technical Report TR-008, Institute for New Generation Computer Technology, May 1983.

[Hirschberg 83] P. S. Hirschberg and D. S. Volper. A parallel solution for the minimum spanning tree problem. In *Seventeenth Annual Conference on Information Sciences and Systems*, pages 680–684, 1983.

[Hoare 78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug 1978.

[Ibarra 86] O. H. Ibarra, S. M. Kim, and M. A. Palis. Designing systolic algorithms using sequential machines. *IEEE Transactions on Computers*, 35(6):531–542, June 1986.

[Inmos 86] Inmos Limited, Bristol, U. K. *Transputer Reference Manual*, 1986.

[Kasami 65] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free analysis. Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass., 1965.

[Kennedy 79] K. Kennedy. Optimization of vector operations in an extended fortran compiler. Research Report RC-7784, IBM, 1979.

[Knuth 65] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[Knuth 68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–146, 1968. see [Knuth 68].

[Knuth 71a] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[Knuth 71b] D. E. Knuth. Semantic of context-free languages, correction. *Mathematical Systems Theory*, 5:95–96, 1971. A correction to [Knuth 68].

[Korenjak 66] A. J. Korenjak and J. E. Hopcroft. Simple deterministic languages. In *Conference Records of the 7th Annual Symposium on Switching and Automata Theory*, pages 36–46. IEEE, 1966.

[Kosaraju 69] S. R. Kosaraju. *Computations on Iterative Automata.* PhD thesis, University of Pennsylvania, 1969.

[Kosaraju 75] S. R. Kosaraju. Speed of recognition of context-free languages by array automata. *SIAM Journal of Computing*, 4(3):331–340, September 1975.

[Kung 79] H. T. Kung and C. E. Leiserson. Systolic arrays for VLSI. In *Caltech Conference on VLSI*, Jan 1979.

[Lewis 68] P. M. II Lewis and R. E. Stearns. Syntax directed transduction. *Journal of the ACM*, 15(3):464–488, 1968.

[Lincoln 70] N. Lincoln. Parallel programming techniques for compilers. *SIGPLAN Notices*, 5(10), 1970.

[Lipkie 79] D. E. Lipkie. *A Compiler Design for Multiple Independent Processor Computers.* PhD thesis, University of Washington, 1979.

[McCarthy 60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Communications of the ACM*, 3(4):184–195, 1960.

[Mead 80] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[Medina 81] R. Medina-Mora and P. H. Feiler. An incremental programming environment. *IEEE Transactions on Software-Engineering*, 7(5):472–482, Sep 1981.

[Menabrea 61] L. F. General Menabrea. Sketch of the analytical engine invented by Charles Babbage. In P. Morrison and E. Morrison, editors, *Charles Babbage and his Calculating Engines: Selected Writings by Charles Babbage et al.*, page 244. Dover, New York, 1961.

[Mickunas 78] M. D. Mickunas and J. A. Modry. Automatic error recovery for LR parsers. *Communications of the ACM*, 21(6):459–465, Jun 1978.

[Miller 84] R. Miller and Q. F. Stout. Computational geometry on a mesh-connected computer. In *1984 International Conference on Parallel Processing*, 1984.

[Milner 80] Robin Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science. Springer-Verlag, Berlin, New York, 1980.

[Miranker 71] W. L. Miranker. A survey of parallelism in numerical analysis. *SIAM*, 13:524-547, 1971.

[Parkinson 88] D. Parkinson, D. J. Hunt, and K. S. MacQueen. The AMT DAP 500. In IEEE Computer Society Press, editor, *Proceedings of the Thirty-Third IEEE Computer Society International Conference, San Francisco, California*, pages 196–199, Washington, D. C., Feb 1988. IEEE Computer Society.

[Paul 75] G. Paul and M. W. Wilson. The VECTRAN language: An experimental language for vector/matrix array processing. Report 6320-3334, IBM Palo Alto Scientific Center, Aug 1975. (a vectorization compiler).

[Poole 74] J. M. Poole Jr. and R. G. Voight. Numerical algorithms for parallel and vector computers: an annotated bibliography. *Computing Review*, 9:61–102, 1974.

[Randell 82] Brian Randell, editor. *The Origins of Digital Computers: Selected Papers.* Springer-Verlag, New York, Berlin, 1982.

[Reddaway 79] S. F. Reddaway. The DAP approach. In C. R. Jesshope and R. W. Hockney, editors, *Infotech State of the Art Report: Supercomputers*, pages 311–329. Infotech International Ltd., Maindenhead, 1979.

[Reps 83] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependant analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, Jul 1983.

[Reynolds 65] J. C. Reynolds. An introduction to the COGENT programming system. In *Proceedings of the ACM National Conference*, page 422. ACM, 1965.

[Schell 79] R. M. Schell Jr. *Methods for Constructing Parallel Compilers for use in a Multiprocessor Environment.* PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[Schorre 64] D. V. Schorre. META II, a syntax oriented compiler writing language. In *Proceedings of the ACM 19th National Conference*, pages D1.3-1–D1.3-11. ACM, 1964.

[Shaw 80] D. E. Shaw. *Knowledge-based retrieval on a Relational Database Machine.* PhD thesis, Dept. of Computer Science, Stanford University, 1980.

[Smith 71] A. R. Smith III. Two-dimensional formal languages and pattern recognition by cellular automata. In IEEE, editor, *IEEE Switching and Automata Theory Conference*, pages 144–152. IEEE, 1971.

[Snyder 82] L. Snyder. Introduction to the configurable, highly parallel computer. *Computer*, 15(1):47–56, January 1982.

[Steele 75] G. L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, Sep 1975.

[Strassen 69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, (13):354–356, 1969.

[Tappert 78] C. C. Tappert. Memory and time improvements in a dynamic programming algorithm for matching speech patterns. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 26(6):583–586, Dec 1978.

[Teitelbaum 81] T. Teitelbaum, T. Reps, and S. Horowitz. The Cornell Program Synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, Sep 1981.

[Thompson 77] C. D. Thompson and H.-T. Kung. Sorting on a mesh connected parallel computer. *Communications of the ACM*, 20:263–271, 1977.

[Thompson 84] H. Thompson. Speech transcription: An incremental interactive approach. In T. O'Shea, editor, *Advances in Artificial Intelligence*, pages 697–704. North-Holland, 1984.

[Valiant 75] L. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308–315, 1975.

[Wikstrom 87] Ake Wikstrom. *Functional Programming using Standard ML*. Prentice-Hall Series in Computer Science. Prentice-Hall International, 1987.

[Winograd 83] T. Winograd. Charts and active chart parser. In *Language as a Cognitive Process, Syntax*, pages 116–127. Addison-Wesley, 1983.

[Wirth 66] N. Wirth and H. Weber. EULER— a generalization of ALGOL and its formal definition. *Communication of the ACM*, 9(2):89–99, 1966.

[Wood 70] D. Wood. The theory of left factored languages. *Computer Journal*, 13(1):55–62, 1970.

[Younger 67] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

[Zosel 73] M. Zosel. A parallel approach to compilation. In *Symposium on the Principles of Programming Language*, pages 59–70. ACM, 1973.