

# **IMPACT**

Parallel Natural Language Interfaces

## Survey of Parallel Context-Free Parsing Techniques

M. P. van Lohuizen

**IMPACT-NLI-1997-1**



# IMPACT

Parallel Natural Language Interfaces

Survey of Parallel Context-Free Parsing Techniques

Published and produced by:  
Faculty of Information Technology and Systems,  
Department of Technical Mathematics and Informatics,  
Parallel and Distributed Systems Section.  
Delft University of Technology  
Zuidplantsoen 4  
2628 BZ Delft  
The Netherlands

Information about the Parallel and Distributed Systems Section:  
<http://pds.twi.tudelft.nl/>

Submitted to: ING Bank, leader of the HPCN project IMPACT.

Uitgebracht aan: ING Bank, projectleider HPCN project IMPACT.

Delft University of Technology  
Parallel and Distributed Systems Report Series

Survey of Parallel Context-Free Parsing Techniques

M. P. van Lohuizen

report number PDS-1997-003

**PDS**

ISSN 1387-2109

Published and produced by:  
Parallel and Distributed Systems Section  
Faculty of Information Technology and Systems Department of Technical Mathematics  
and Informatics  
Delft University of Technology  
Zuidplantsoen 4  
2628 BZ Delft  
The Netherlands

Information about Parallel and Distributed Systems Report Series:  
[reports@pds.twi.tudelft.nl](mailto:reports@pds.twi.tudelft.nl)

Information about Parallel and Distributed Systems Section:  
<http://pds.twi.tudelft.nl/>

# Preface

This report describes research done in the context of a subproject of the HPCN project IMPACT. The IMPACT project is headed by the ING bank and is founded by the organization for High Performance Computing and Networking (HPCN). The aim of the specific subproject, in the context of which this report has been written, is to develop (techniques for) natural language interfaces to information resources, focusing on the use of high-performance computers to achieve acceptable response times. This report is part of the “Parallel Parsing I” research topic.





# Contents

<b>Preface</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Basics</b>	<b>3</b>
2.1 Context-Free Grammars . . . . .	3
2.2 Requirements for Parallel Parsers . . . . .	4
2.3 Requirements for Parsers for NLP . . . . .	4
2.4 Types of Parallelism . . . . .	5
<b>3 Parsing Methods</b>	<b>9</b>
3.1 Introduction . . . . .	9
3.1.1 Parsing Schemata . . . . .	10
3.1.2 Generalization . . . . .	11
3.1.3 Filtering . . . . .	12
3.1.4 Parallel Implementation . . . . .	13
3.2 CYK Parsing . . . . .	14
3.3 Rytter and OCYK Parsing . . . . .	14
3.4 Earley Parsing . . . . .	16
3.5 Double Dotted Parsing . . . . .	18
3.6 Left- and Head-corner Parsing . . . . .	20
<b>4 Parallel Parsing Strategies</b>	<b>25</b>
4.1 From One to Many Traditional Parsers . . . . .	25
4.1.1 Parsing Separate Parts of the Input String . . . . .	26
4.1.2 Parallel Processing in case of Non-Determinism . . . . .	26
4.2 Parallel Chart Parsing . . . . .	27
4.2.1 Basic Chart Parsing . . . . .	27
4.2.2 String Chart Parsing . . . . .	28
4.2.3 Tabular Chart Parsing . . . . .	30
4.3 Translating Grammar Rules into Process Configurations . . . . .	33
4.4 Connectionist Parsing Algorithms . . . . .	36
4.4.1 Connectionist CYK . . . . .	37
4.4.2 Other Connectionist Approaches . . . . .	38
4.5 Reducing the Parsing Problem to Other Problems . . . . .	38

4.5.1	Parsing as Matrix-Multiplication . . . . .	38
4.5.2	Context-Free Parsing as Parallel Logic Programming . . . . .	40
<b>5</b>	<b>Suggestions for Future Research</b>	<b>43</b>
5.1	Other Approaches to Parsing . . . . .	43
5.2	Existing Natural Language Processing Systems . . . . .	44
5.3	Context-Free Parsing and the IMPACT Project . . . . .	46
5.4	Directions for Further Research . . . . .	46
	<b>References</b>	<b>47</b>

# Chapter 1

## Introduction

The demand for increasingly complex natural language processing systems has stimulated the research of using parallel architectures for such systems. The belief in the feasibility of efficient parallel natural language processing has been encouraged by the observation that humans themselves process at least some parts of language in parallel (see note [NH96]).

Parsers, which can be used for both syntactic and semantic processing, are one of the essential components of natural language processing systems. At this time, parsers for attribute-value grammars seem most appropriate for this task. Such parsers can cover a wide variety of different formalisms that are used today for natural language processing. Therefore, our research will initially focus on the implementation of practical parallel parsers for such grammars.

Basically, attribute-value grammars consist of a context-free backbone augmented with term unification. Both aspects appear in parsers for such grammars as separate components. Therefore, making parallel implementations of parsers for such grammars requires (at least at first) separate treatment of those components. In this report we will focus on the parallelization of one of these components: the context-free backbone. Parallelization of parsers for full attribute-value grammars will be the topic of further research.

The ultimate objective of our research is to obtain a dialog system—and to gain knowledge about techniques for making such a system—that can be used to query a relational database. Such systems allow a user to engage in a dialog and ask a question that the system will have to translate to a query language, e.g. SQL. Dialog systems with workable response times are of main interest, hence the research will initially focus on parallelization techniques.

There is a large gap between the theory and practice of natural language processing. Linguistic and computer science theorists attach great value to properties like completeness of the parser. In reality, the prevalent requirement is that a dialog system can engage in an efficient and co-operative dialogue ([vdBAK<sup>+</sup>96]). In other words: it has to work. For example, [vdBAK<sup>+</sup>96] notes a dialog interface is successful if a user feels comfortable with the dialog interface. Even if the dialog interface uses a restricted language, as long as the user is comfortable with it, it is an acceptable system. Nevertheless, efficient parsers are still of great importance. Initially, our research will mainly

focus on the development of efficient *parallel* parsers.

There are several circumstantial aspects that are of influence on our research. First, previous research at the university has resulted in a Dutch grammar. Reusing such a grammar saves a great deal of effort. On the other hand, reuse may also limit the possible parsing techniques from which one can choose.

Second, the dialog system will most likely be used in an environment which includes an n-CUBE<sup>TM</sup>. Therefore, effort will be needed in the research of parallel parsing using a hypercube architecture.

The rest of the report is structured as follows. Chapter 2 covers some basic notions of parallel context-free parsing in the context of natural language processing and some methodologies for parallel processing. Chapter 3 gives an overview of different parsing methods within the framework of parsing schemata. Within this framework, parsers can be specified independent of their algorithmic aspects. Chapter 4 discusses different approaches to parallelizing parsing algorithms. Many of such approaches are more or less independent of the underlying parsing method. So, whereas chapter 3 discusses some parsing methods in detail, chapter 4 discusses some approaches to the implementation of parallel parsing algorithms. Finally, chapter 5 presents some conclusions and suggestions for further research.

## Chapter 2

# Basics

As mentioned in the introduction, context-free parsers are one of the two essential components of parsers for attribute-value grammars. With attribute-value grammars, non-terminals are allowed to be assigned attribute value pairs. The parsing of such grammars requires term unification, in addition to mechanisms required for context-free parsing. The context-free ‘backbone’ of parsers for AV grammars, however, remains unchanged. In this report, we will limit our attention to parallel context-free parsing. Parallel term unification will be subject of further research.

We continue this chapter with some terminology and notational conventions concerning CFGs. In sections 2.2 and 2.3, we will discuss some of the requirements and points of attention in the development of parallel parsers and parsers for natural language processing, respectively. In section 2.4, we discuss several general approaches to parallelism.

### 2.1 Context-Free Grammars

A *context-free grammar*  $G \in \mathcal{CFG}$  is defined as a 4-tuple  $\langle N, \Sigma, P, S \rangle$ , where  $N$  is the set of non-terminal symbols,  $\Sigma$  the set of terminal symbols, with  $N \cap \Sigma = \emptyset$ ,  $P$  the set of production rules of the form  $N \Rightarrow (N \cup \Sigma)^*$ , and  $S$  the start symbol. In addition, we will often write  $V$  for  $N \cup \Sigma$ .

We will use  $A, B, \dots$  to denote any non-terminal in  $N$  and we will use  $a, b, \dots$  to denote any terminal in  $\Sigma$ . Occasionally, we will use  $X, Y, \dots$  to denote any element in  $V = N \cup \Sigma$ . An arbitrary string in  $V^*$  is denoted by  $\alpha, \beta, \dots$ . We denote an arbitrary string in  $\Sigma^*$  (an input string) of length  $n$  by  $a_1 \dots a_n$ . Finally, the empty string is denoted by  $\epsilon$ .

We will often consider a subclass of context-free grammars called the Chomsky Normal Form. A grammar  $G$  is an element of  $\mathcal{CNF}$  iff all  $p \in P$  are either of the form  $A \rightarrow a$  or  $A \rightarrow BC$ . Every context-free grammar  $G$  for which  $\epsilon \notin L(G)$  can be rewritten into an equivalent grammar in  $\mathcal{CNF}$ .

A recognizer for a grammar  $G$  can determine whether an arbitrary string is an element of the language  $L(G)$  produced by that grammar. In addition, a parser constructs a parse tree in the case a string is recognized. Obviously, parsing a string is a more

difficult task than merely recognizing a string. Nevertheless, we will often only consider recognizers when referring to parsers, because it is often straightforward to convert a recognizer into a parser or to obtain a parse tree from the data structures that were already built by the recognizer.

Besides the desirable characteristics of soundness and completeness, there are several design goals for parallel parsers and for parsers for natural language processing in particular that need be mentioned.

## 2.2 Requirements for Parallel Parsers

In the field of parallel algorithms we can distinguish slow- and fast-parallelism. Slow-parallelism aims at achieving speedups while keeping the processor-time product constant, whereas fast-parallelism focuses on algorithms that run in polylogarithmic time. In addition, a fast parallel algorithm is called feasible if it uses not more than a polynomial number of processors. In practice, an algorithm can only be called feasible if it requires no more than a linear or at most a quadratic number of processors.

In the design of practical parallel parsers the following measures are relevant.

**Time Complexity** One can either pursue slow- or fast-parallelism, depending on the processor complexity one wishes to allow.

**Processor Complexity** The number of processors a certain algorithm requires.

**Communication Complexity** Parallel implementations or parsers seem to need extensive communication.

**Space Complexity** The amount of space that is required per processor.

A problem with the design of parallel algorithms is that reducing one of the complexities often has the result of increasing another. For example, optimizing time complexity can result in an increasing processor complexity and hence possibly an increase in communication complexity. Reducing the space complexity can result in an increased communication complexity, because this often requires more sophisticated scheduling policies.

## 2.3 Requirements for Parsers for NLP

Parsers can be used for a wide variety of applications. The most prevalent areas of application are compiling and natural language processing. [dV93a] mentions several differences between applications in these two areas, amongst which the thousands of lines of code against sentences usually not longer than 30 words, the usually small, unambiguous programming languages against highly ambiguous natural language, and the often deep and highly skewed derivation trees of programs against fairly balanced derivation trees in natural languages.

These differences yield differences for the requirements for the parsers. To be useful for practical NLP applications we often want parsers to fulfill more requirements than simply those of efficiency. The following list contains several requirements specific to natural language processing.

**Handling of Ambiguities** A parser should be capable of handling ambiguities, for example, by returning all possible parse trees or finding the most likely parse. An important quality of parsers is the capability of handling ambiguity without much performance loss.

**Robustness** Humans make errors. A parser for natural language processing purposes should be capable of dealing with minor errors, e.g., by finding useful partial parses or determining a parse tree for the most likely intended sentence.

**Class of Grammars** The class of grammars that can be handled by the parser is of interest, because extensions or limitations of context-free grammars can yield improved performance.

If a parser is used on-line, for example within a dialog system, the following abilities will increase its usefulness:

**Incremental parsing** This means that parsing can start before the complete input string is known. This enables parsing to start while the sentence is still being uttered.

**Constant time complexity per word** Constant time complexity per word is desirable, because this prevents temporary performance degradation, which may be annoying to the user.

**Undo capabilities** Undo capabilities will allow a speaker (or better typist) to correct already uttered input. A parser with this capability is capable of reparsing part of the input, without redoing already completed work as much as possible. Undo capabilities usually appear as either backspacing capabilities or full undo capabilities.

Throughout the report, we will mention the conformance to these requirements of any parsing technique whenever it applies.

## 2.4 Types of Parallelism

For completeness, we will provide a succinct taxonomy of parallelization approaches. The overview has large been taken from [AH94]. This taxonomy is based on the von Neumann model of computation, generalizing the model to, e.g., SIMD and MIMD architectures.

### Control-Driven Parallelism

This paradigm conserves the serial nature of programming. Parallelism is introduced either by smart compilers, which can recognize, for example, possible parallelism in loops, or by explicit language constructs, which the programmer inserts in the basically imperative program to indicate possible parallelism.

Because different threads are capable of altering variables that are shared amongst the different processes, side-effects are common. This capability can cause serious difficulties, but also provides powerful means of communication and sharing data structures amongst the processes.

### Demand-Driven Parallelism

The demand-driven paradigm is based on a functional point of view. A program is regarded as a collection of functions. As a consequence, there is no concept of state, program counter, time dependence, or data storage. Side-effects are impossible.

The meaning of any expression is completely specified by the meaning of its subexpressions. This means that parallelism can easily be introduced at the level of subexpressions. All subexpressions can be assigned a separate processor, recursively. Parallelization comes entirely for free. Implementations of this strategy often use lazy evaluation.

### Data-Driven Parallelism

With data-driven parallelism, or data-flow programming, the exact order of dependencies among the single operations is specified. Data flows through a network of operations, where the output of one operation is the input of another. This scheme can be used as both the principal means of design and a paradigm to parallelize control-driven programs.

The essential difference with the control-driven approach is that the programmer indicates which output is another computation's input, instead of specifying which operation should be executed before another. As a bonus, specifying data-dependencies also yields the conditions that need to be satisfied to execute a particular instruction: an instruction may be executed after all its inputs become available.

Both with the demand-driven and with the data-driven approach, the control is specified implicitly. A big difference between the two, though, is that in the data-driven model control flows bottom-up, whereas in the demand-driven model control flows top-down.

### Pattern-Driven Parallelism

With pattern-driven parallelism, an instruction is executed when certain conditions are met. Typical examples of a pattern-driven approach are production systems, which are executed by means of the Match-Select-Act model. A formal approach to these kind of systems can be found in logic programming systems, which are based on first order logic. Since, usually, production system interpreters spend 90% of their time in the Match step, parallelizing the Match step can yield the best performance increase.

We can distinguish three approaches to the parallelization of logic programming systems: OR-parallelism, AND-parallelism, and unification parallelism.

**OR-parallelism** Various solutions to a goal are evaluated by allowing the alternative clauses for a goal to be evaluated in parallel. With pure OR-parallelism, the different parallel processes are independent of each other, and hence no communication between the different processes are needed. In the OR-processes approach, there are several objects that each are responsible for a piece of the program (a subset of the parallel processes of pure OR-parallelism). The processes communicate by sending messages. This introduces overhead, but there is more control over the granularity of the parallelism, and it is easy to mix OR- and AND-processes.



Obviously, OR-parallelism can only occur if the production rules exhibit non-determinism. The limited amount of non-determinism that is encountered in practice, along with performance concerns, has resulted in a shift towards AND-parallelism ([AH94]).

**AND-parallelism** The execution of several goals of a clause in parallel. Contrary to OR-parallelism, the amount of parallelism is independent of the amount of non-determinism. A serious problem that occurs with AND-parallelism, though, is the possibility of conflicts in the case the separate goals share variables. Full AND-parallelism overcomes this problem by simply ignoring conflicts and joining (relational algebra) the results of all the goals, so that only valid combinations remain. This approach, just as with OR-parallelism, produces all possible solutions, and may therefore be time and memory consuming.

The approach for handling conflicts that is given by restricted AND-parallelism is to avoid processing goals in parallel that share unbound variables. [AH94] mentions some efficient implementations of this approach.

A radically different approach to conflict handling is used in stream parallelism. Instead of avoiding conflicts, some form of communication is used to pass values of variables. One approach is to pipeline the goals in a producer-consumer-like fashion. As soon as a producer goal binds a variable, it is passed to its successor, which can then start processing itself. [AH94] mentions some implementations of this approach. Another approach is called cooperative parallelism, which allows goals to be executed in parallel working on a single solution, using bidirectional communication.

**Unification parallelism** Unification parallelism applies to the unification process as it occurs in logic programming systems. With the unification parallelism approach, the unification of clauses in the database with the goal clause, including terms, is parallelized. It is strongly believed that the unification problem is log-space complete for **P**, which means the problem is inherently sequential (see, e.g., [DKM84] and [Yas84]). Nevertheless, parallelizing term matching, obviously a subproblem of unification, seems to be profitable. Amongst several others, [AH94] mentions an implementation that requires  $O(n^3)$  processors, running in  $O(\log_2 n)$  time.

For an in depth coverage see [Kac90].

### Object-Oriented Parallelism

With the object-oriented approach to parallelism, objects (as in object-oriented programming) are considered as the units of parallelization. Objects do not have any shared variables, and their only way to communicate is by sending messages (methods). This scheme closely corresponds to concurrent programming, where objects can be seen as processes and messages can be seen as interprocess communication. The focus of this approach is on introducing parallelism by means of an effective distribution of knowledge among multiple computing agents and a well designed communication scheme.

### Spreading Activation Parallelism

Spreading activation parallelism, or more commonly, connectionism, is also common in natural language processing. [Nij91] provides an overview of the different approaches that have been pursued. See also section [4.4](#).

## Chapter 3

# Parsing Methods

Different parsing methods often seem related, although from the specific algorithms it is not clear how. Parsing schemata allow us to specify parsing methods at a level between grammars and algorithms. This enables us to capture the essence of different parsing techniques.

In section 3.1, we will give a concise overview of parsing schemata, along with some of the associated theory. In addition, we will identify which variation of characteristics of parsing methods affect the opportunities for parallel processing. The contents of this chapter have largely been taken from [SN97]; for a full formal description of parsing schemata, we refer to [Sik93b].

In the remainder of the section, we will list a wide variety of parsing methods, and mention some interesting relations between them.

### 3.1 Introduction

Parsing schemata were originally developed to provide a unifying approach to context-free parsing. The framework, however, can also be applied to parsing methods that deal with grammars beyond the context-free category. This section covers some of the parsing schemata theory, but is by no means complete. Its purpose is solely to be a summary. Consequently, definitions may sometimes seem incomplete.

The subsections about filtering and generalization discuss the possibility of identifying relations between parsing schemata of different parsing methods. The theory gives a precise definition of what these generalizations and filters are. Once it is known how one parsing method transforms into another, it is sometimes possible to apply similar transformations to other methods. This allows, for example, desirable characteristics of one method to be transferred to another, or a well-balanced average of two methods to be obtained.

Finally, the subsection about parallel implementation addresses some parallel parsing aspects in relation to parsing schemata.

### 3.1.1 Parsing Schemata

Parsing schemata are inspired by the “item-based” approach to parsing. According to this point of view, recognizing is a process of deducing a final set of items (possibly representing complete parse trees) from an initial set of items (the hypotheses) by means of a set of deduction rules. Different parsing methods use different items (item domains), deduction rules, and sets of initial items. Most parsing methods, including Earley-, LR-, and unification-style parsing methods, can effectively be interpreted as item-based parsers.

Let us first define a parsing system, which is defined for a specific grammar and input string.

**Definition 3.1 (Parsing System).** A parsing system  $\mathbb{P}$  for some grammar  $G$  and input-string  $a_1 \dots a_n$  is a triple  $\mathbb{P} = \langle \mathcal{I}, H, D \rangle$ , in which

- $\mathcal{I}$  is the domain or item set of  $\mathbb{P}$ , which specifies the allowed items. (The details of the syntax of items may be different per schema.)
- $H$  is a finite set of initial items, or hypotheses. ( $H$  need not be a subset of  $\mathcal{I}$ .)
- $D \subseteq \wp_{fin}(H \cup \mathcal{I}) \times \mathcal{I}$  is a set of production steps, where  $\wp_{fin}(H \cup \mathcal{I})$  represents all finite elements in the power-set of  $(H \cup \mathcal{I})$ .

Parsing systems define a parsing method for a specific grammar  $G$  and input string  $a_1 \dots a_n$ . Uninstantiated parsing systems are defined for an arbitrary input string.

**Definition 3.2 (Uninstantiated Parsing System).** An uninstantiated parsing system for a grammar  $G$  is a function that assigns a parsing system to any  $a_1 \dots a_n \in \Sigma^*$ . A uninstantiated parsing system is defined by a triple  $\langle \mathcal{I}, \mathcal{H}, D \rangle$ , where  $\mathcal{H}$  is a function that assigns a set of hypotheses to a string  $a_1 \dots a_n \in \Sigma^*$ .

A function  $\mathcal{H}$  is usually defined as

$$\mathcal{H}(a_1 \dots a_n) = \{[a, i-1, i] \mid a = a_i \wedge 1 \leq i \leq n\}. \quad (3.1)$$

**Definition 3.3 (Parsing Schema).** A parsing schema for some (sub)class of context-free grammars  $\mathcal{CG} \subseteq \mathcal{CFG}$  is a function that assigns an uninstantiated parsing system to any grammar  $G \in \mathcal{CG}$ .

In [SN97], parsing schemata are always specified by means of a parsing system, after which this parsing system is generalized for all grammars and input strings. In the remainder of this section, we will often simply use the parsing system notation for parsing schemata, by which we implicitly assume that the definition holds for all  $G \in \mathcal{CG}$ , for some class of grammars  $\mathcal{CG}$ , and all input strings  $a_1 \dots a_n \in \Sigma^*$ .

The following definitions are needed to specify generalizations and filters in the next two subsections.

**Definition 3.4 (Inference Relation  $\vdash$ ).** For a given  $\mathbb{P} = \langle \mathcal{I}, H, D \rangle$ , the relation  $\vdash \subseteq \wp_{fin}(H \cup \mathcal{I}) \times \mathcal{I}$  is defined by

$$Y \vdash \xi \text{ if } (Y', \xi) \in D \text{ for some } Y' \subseteq Y. \quad (3.2)$$

Often,  $Y \vdash \xi$  is used, instead of  $\{\eta_1 \dots \eta_n\} \vdash \xi$ , if  $Y = \{\eta_1 \dots \eta_n\}$ .

**Definition 3.5 (Deduction Sequence).** A deduction sequence for a parsing system  $\mathbb{P} = \langle \mathcal{I}, \mathcal{H}, \mathcal{D} \rangle$  is a pair  $(Y; \xi_1, \dots, \xi_{i-1}) \in \wp_{fin}(\mathcal{H} \cup \mathcal{I}) \times \mathcal{I}^+$ , such that  $Y \cup \{\xi_1 \dots \xi_{i-1}\} \vdash \xi_i$ , for  $1 \leq i \leq j$ .

We will usually write  $Y \vdash \xi_1 \dots \xi_j$  for  $(Y; \xi_1, \dots, \xi_j)$ .

**Definition 3.6 (Set of Deduction Sequences  $\Delta$ ).** The set of deduction sequences  $\Delta \subseteq \wp_{fin}(\mathcal{H} \cup \mathcal{I}) \times \mathcal{I}^+$  for  $\mathbb{P} = \langle \mathcal{I}, \mathcal{H}, \mathcal{D} \rangle$  is defined by

$$\Delta = \{(Y; \xi_1, \dots, \xi_j) \in \wp_{fin}(\mathcal{H} \cup \mathcal{I}) \times \mathcal{I}^+ \mid Y \vdash \xi_1 \vdash \dots \vdash \xi_j\}.$$

Armed with this knowledge, we can take a closer look at generalizations and filtering.

### 3.1.2 Generalization

Generalization can qualitatively improve parsing methods, by allowing a more fine-grained look at the parsing process. The disadvantage is that the number of steps to be performed increases.

Generalizations of parsing methods take the form of either refinements into a more detailed parsing schema, or extensions to a larger class of grammars. [SN97] distinguishes three type of basic refinements: item refinements, step refinements, and extensions. For the next definitions we will use parsing systems  $\mathbb{P}_i = \langle \mathcal{I}_i, \mathcal{H}_i, \mathcal{D}_i \rangle$ , which are associated with the relations  $\vdash$  and  $\vdash^*$  based on  $\mathcal{D}_i$ .

With an item refinement of a parsing schema  $\mathbf{P}_1$  into  $\mathbf{P}_2$ , a single item of  $\mathbf{P}_1$  is broken down into multiple items in  $\mathbf{P}_2$ , and the set of reduction steps is adapted accordingly.

In the definition of item refinements, we make use of an item mapping. An item mapping  $f : \mathcal{I}_2 \rightarrow \mathcal{I}_1$  simply maps items of  $\mathbb{P}_2$  to items of  $\mathbb{P}_1$ . Its definition can be extended to sets of items as follows:

$$f(Y) = \{\xi \in \mathcal{I}_1 \mid \exists \eta \in Y : f(\eta) = \xi\}.$$

Similarly, we can extend its definition to a function  $f : \mathcal{I}_2 \cup \mathcal{H} \rightarrow \mathcal{I}_1 \cup \mathcal{H}$ , where we let  $f(h) = h$ , for all  $h \in \mathcal{H}$ . We can now define  $f$  for the domain of deduction steps by letting

$$f(\eta_1, \dots, \eta_k \vdash \xi) = f(\eta_1), \dots, f(\eta_k) \vdash f(\xi).$$

Similarly, we can extend  $f$  to deduction sequences, sets of deduction steps, and sets of deduction sequences.

**Definition 3.7 (Item Refinement).** The relation  $\mathbb{P}_1 \xrightarrow{\text{ir}} \mathbb{P}_2$  holds if there is an item mapping  $f : \mathcal{I}_2 \rightarrow \mathcal{I}_1$  such that

1.  $\mathcal{I}_1 = f(\mathcal{I}_2)$ ,
2.  $\Delta_1 = f(\Delta_2)$ .

Furthermore, the relation  $\mathbf{P}_1 \xrightarrow{\text{ir}} \mathbf{P}_2$  holds if  $\mathbf{P}_1(G)(a_1 \dots a_n) \xrightarrow{\text{ir}} \mathbf{P}_2(G)(a_1 \dots a_n)$  for all  $\mathbf{P}_1$  and  $\mathbf{P}_2$  for a class of grammars  $\mathcal{CG}$ ,  $G \in \mathcal{CG}$ , and  $a_1 \dots a_n$ .

Condition 1 ensures that all items of the system to be refined are preserved; condition 2 ensures that all deduction sequences of  $\mathbb{P}_1$  have their equivalent in the refined system.

With a step refinement, a single deduction step in  $\mathbf{P}_1$  is broken down into a sequence of reduction steps in  $\mathbf{P}_2$ , and new items are introduced if these are required to store intermediate results. Step refinement is defined in a similar fashion as item refinement.

**Definition 3.8 (Step Refinement).** Similar as item refinement, but now  $\mathbb{P}_1 \xrightarrow{\text{sr}} \mathbb{P}_2$  holds if

1.  $\mathcal{I}_1 \subseteq \mathcal{I}_2$ ,
2.  $\vdash_1^* \subseteq \vdash_2^*$ .

For condition 2 it suffices to take  $\mathcal{D}_1 \subseteq \vdash_2^*$ , which means: a single deduction step in  $\mathbb{P}_1$  is emulated by a sequence of deduction steps in  $\mathbb{P}_2$ . The choice for the given condition is merely because of symmetry.

These two generalizations together are called refinement:

**Definition 3.9 (Refinement).** For any  $\mathbf{P}_1$  and  $\mathbf{P}_2$  for a class of grammars  $\mathcal{CG}$ , relation  $\mathbf{P}_1 \xrightarrow{\text{ref}} \mathbf{P}_2$  holds if there is a parsing schema  $\mathbf{P}'$  such that  $\mathbf{P}_1 \xrightarrow{\text{ir}} \mathbf{P}' \xrightarrow{\text{sr}} \mathbf{P}_2$ .

A parsing schema  $\mathbf{P}_2$  is called an extension of  $\mathbf{P}_1$ , if it is defined for a larger class of grammars.

**Definition 3.10 (Extension).** Let  $\mathbf{P}_1$  be a parsing schema for a class of grammars  $\mathcal{CG}_1$ , and let  $\mathbf{P}_2$  be a parsing schema for a class of grammars  $\mathcal{CG}_2 \supseteq \mathcal{CG}_1$ . Then  $\mathbf{P}_1 \xrightarrow{\text{ext}} \mathbf{P}_2$  holds if  $\mathbf{P}_1(G) = \mathbf{P}_2(G)$  for all  $G \in \mathcal{CG}_1$ .

**Definition 3.11 (Generalization).** Let  $\mathbf{P}_1$  be a parsing schema for a class of grammars  $\mathcal{CG}_1$ , and let  $\mathbf{P}_2$  be a parsing schema for a class of grammars  $\mathcal{CG}_2 \supseteq \mathcal{CG}_1$ . Then relation  $\mathbf{P}_1 \xrightarrow{\text{gen}} \mathbf{P}_2$  holds if there is a parsing schema  $\mathbf{P}'$  such that  $\mathbf{P}_1 \xrightarrow{\text{ref}} \mathbf{P}' \xrightarrow{\text{ext}} \mathbf{P}_2$ .

### 3.1.3 Filtering

Filtering can be regarded as the converse of generalization. The overall goal of filtering is usually to reduce the number of steps that is needed to complete the parsing, which can be accomplished by both reducing the number of items and reducing the number of deduction steps. It is often possible to discard items or deduction steps without weakening the generality of the parser. Such optimization can lead to more efficient algorithms, but also to more complicated descriptions of the parsing schema.

[SN97] distinguishes also three types of filters: static filtering, dynamic filtering, and step contraction. With static filtering redundant parts of the schema are simply discarded. We define static filtering as follows.

**Definition 3.12 (Static Filtering).** The relation  $\mathbb{P}_1 \xrightarrow{\text{sf}} \mathbb{P}_2$  holds if

1.  $\mathcal{I}_1 \supseteq \mathcal{I}_2$ ,
2.  $\mathcal{D}_1 \supseteq \mathcal{D}_2$ .

Furthermore, the relation  $\mathbf{1} \xrightarrow{\text{sf}} \mathbf{2}$  holds if  $\mathbf{P}_1(G)(a_1 \dots a_n) \xrightarrow{\text{sf}} \mathbf{P}_2(G)(a_1 \dots a_n)$  for all  $\mathbf{P}_1$  and  $\mathbf{P}_2$  for a class of grammars  $\mathcal{CG}$ ,  $G \in \mathcal{CG}$ , and  $a_1 \dots a_n$ .

Dynamic filtering allows taking context information into account. For example, if we know that a partial parse, represented by an item  $\xi$ , is only correct if  $\zeta$  is a valid item, we can replace the reduction steps  $\eta_1, \dots, \eta_k \vdash \xi$  by  $\eta_1, \dots, \eta_k, \zeta \vdash \xi$ . An example of dynamic filtering is *look-ahead*. Dynamic filtering is defined as follows.

**Definition 3.13 (Dynamic Filtering).** Similar to static filtering, but here  $\mathbb{P}_1 \xrightarrow{\text{df}} \mathbb{P}_2$  holds if

1.  $\mathcal{I}_1 \supseteq \mathcal{I}_2$ ,
2.  $\vdash_1 \supseteq \vdash_2$ .

Static filtering and dynamic filtering relate to each other in the following way. Static filters only take grammar structure into account. Since it is independent of the input sentence, optimizations can be carried out at compile time. Dynamic filtering, on the other hand, may use the input sentence at hand to filter more items. Hence, dynamic filtering is inherently run-time.

Finally, step contraction means the replacement of sequences of reduction steps by single reduction steps. It is the most powerful of the filtering methods. It is also the inverse of step refinement.

**Definition 3.14 (Step Contraction).** Similar to static filtering, but here  $\mathbb{P}_1 \xrightarrow{\text{sc}} \mathbb{P}_2$  holds if

1.  $\mathcal{I}_1 \supseteq \mathcal{I}_2$ ,
2.  $\vdash_1^* \supseteq \vdash_2^*$ .

### 3.1.4 Parallel Implementation

Finally, we would like to make some remarks concerning parallel implementations of parsing methods. Usually, bottom-up parsers are considered more suitable for parallelization than top-down parsers. Top-down usually requires a centralized control that thwarts parallel execution.

Dynamic filtering usually has the effect of making a parsing method more top-down. Although dynamic filtering may seem to improve the performance at first, it can also reduce the possibilities for parallel processing. By using, for example, the predict deduction step, both the possible computation order of cells is restricted and a more complex communication pattern of processors is needed. This phenomenon is discussed in more detail in section 4.2.3.

It would be interesting to investigate for what reasons some parsing methods are better suited for the implementation of parallel parsers than others.

### 3.2 CYK Parsing

The parsing schema **CYK** is defined by a parsing system  $\mathbb{P}_{\text{CYK}}$ , for all  $G \in \mathcal{CNF}$  and  $a_1, \dots, a_n \in \Sigma^*$ . The class of grammars  $\mathcal{CNF}$  is a subclass of  $\mathcal{CFG}$  which is restricted to the so-called Chomsky Normal Form. Since all grammars in  $\mathcal{CFG}$  can be transformed to a grammar in  $\mathcal{CNF}$ , and because of its simplicity, **CYK** is still a much used parsing method. **CYK** can be defined as follows.

**Parsing Schema 3.1 (CYK).**

$$\begin{aligned} \mathcal{I}_{\text{CYK}} &= \{[A, i, j] \mid A \in N \wedge 0 \leq i < j\} \\ \mathcal{H}_{\text{CYK}} &= \{[a, i-1, i] \mid a = a_i \wedge 1 \leq i \leq n\} \\ \mathcal{D}^{(1)} &= \{[a, i-1, i] \vdash [A, i-1, i] \mid A \rightarrow a \in P\} \\ \mathcal{D}^{(2)} &= \{[B, i, j], [C, j, k] \vdash [A, i, k] \mid A \rightarrow BC \in P\} \\ \mathcal{D}_{\text{CYK}} &= \mathcal{D}^{(1)} \cup \mathcal{D}^{(2)} \end{aligned}$$

### 3.3 Rytter and OCYK Parsing

One of the factors that influences the time complexity of many parsers is the depth of the parse tree. The depth of a parse tree can vary from  $\log_2 n$  to  $n$ . Rytter's parser has the interesting property that it can parse in  $O(\log_2 n)$  time using  $O(n^6)$  processors, independent of the depth of the parse tree.

Rytter's parsing method is a step refinement of **CYK**. The step refinement ensures the existence of a balanced parse tree. This is accomplished by introducing many redundancies, which explains the large number of processors that is required.

Since Rytter is a step refinement of **CYK**, it is only defined for  $\mathcal{CNF}$ . It is possible, however, to apply a similar step refinement to any item-based parser. Under the right circumstances, a new parallel algorithm for the resulting parsing methods can parse in polylogarithmic time, provided that a sufficient number of processors is available.

Rytter refines **CYK** as follows:

$$[B, i, j], [C, j, k] \vdash [A, i, k]$$

refines to

$$\begin{aligned} &[B, i, j] \vdash [A, i, k; C, j, k] \\ &[A, i, k; C, j, k], [C, j, k] \vdash [A, i, k] \end{aligned}$$

The complete schema becomes

**Parsing Schema 3.2 (Rytter).**

$$\mathcal{I}^{\text{recognized}} = \{[A, i, j] \mid A \in N \wedge 0 \leq i < j\}$$



$$\begin{aligned}
\mathcal{I}^{\text{conditional}} &= \{[A, h, k; B, i, j] \mid [A, h, k] \in \mathcal{I}^{\text{recognized}} \wedge [B, i, j] \in \mathcal{I}^{\text{recognized}} \\
&\quad \wedge h \leq i < j \leq k \wedge (h \neq i \vee j \neq k)\} \\
\mathcal{I}_{\text{Rytter}} &= \mathcal{I}^{\text{recognized}} \cup \mathcal{I}^{\text{conditional}} \\
D^{\text{Init}} &= \{[a, i-1, i] \vdash [A, i-1, i] \mid A \rightarrow a \in P\} \\
D^{\text{Activate1}} &= \{[B, i, j] \vdash [A, i, k; C, j, k] \mid A \rightarrow AB \in P\} \\
D^{\text{Activate2}} &= \{[C, j, k] \vdash [A, i, k; B, i, j] \mid A \rightarrow AB \in P\} \\
D^{\text{Combine}} &= \{[A, h, m; , B, i, l], [B, i, l; C, j, k] \vdash [A, h, m; C, j, k]\} \\
D^{\text{Pebble}} &= \{[A, h, k; B, i, j], [B, i, j] \vdash [A, h, k]\} \\
D_{\text{Rytter}} &= D^{\text{Init}} \cup D^{\text{Activate1}} \cup D^{\text{Activate2}} \cup D^{\text{Combine}} \cup D^{\text{Pebble}}
\end{aligned}$$

Items of the form  $[A, i, j]$  represent the recognition of a part of the input string  $a_i \dots a_j$  for a non-terminal  $A$ . Items of the form  $[A, h, k; B, i, j]$  represent the recognition of a part of the input string  $a_h \dots a_k$  for a non-terminal  $A$ , except for the part  $a_i \dots a_j$ , which still needs to be filled with a substring recognized by  $B \Rightarrow^* a_i \dots a_j$ .

There is one interesting parsing method in between **CYK** and **Rytter**, which is particularly well-suited for parallel on-line parsing. With on-line parsing, symbols arrive from left to right. This gives rise to the idea of restricting Rytter items to allow only open parts on the right. This radically reduces the number of possible items.

The resulting parsing method, **OCYK** (see [Sik93a]), is given in the following schema.

### Parsing Schema 3.3 (OCYK).

$$\begin{aligned}
\mathcal{I}^{\text{recognized}} &= \{[A, i, j] \mid A \in N \wedge 0 \leq i < j\} \\
\mathcal{I}^{\text{conditional}} &= \{[A, h, k; B] \mid A, B \in N \wedge 0 \leq i < j\} \\
\mathcal{I}_{\text{OCYK}} &= \mathcal{I}^{\text{recognized}} \cup \mathcal{I}^{\text{conditional}} \\
D^{\text{Init}} &= \{[a, i-1, i] \vdash [A, i-1, i] \mid A \rightarrow a \in P\} \\
D^{\text{Propose}} &= \{[B, i, j] \vdash [A, i, j; C] \mid A \rightarrow AB \in P\} \\
D^{\text{Combine}} &= \{[A, i, j; B], [B, j, k; C] \vdash [A, i, k; C]\} \\
D^{\text{Recognize}} &= \{[A, i, j; B], [B, j, k] \vdash [A, i, k]\} \\
D_{\text{OCYK}} &= D^{\text{Init}} \cup D^{\text{Propose}} \cup D^{\text{Combine}} \cup D^{\text{Recognize}}
\end{aligned}$$

The relation  $\mathbf{CYK} \xrightarrow{\text{sf}} \mathbf{OCYK} \xrightarrow{\text{sf}} \mathbf{Rytter}$  holds. In [Sik93a] an algorithm based on this parsing schema is presented that can parse in  $O(1)$  per word, using  $O(n^2)$  processors.

### 3.4 Earley Parsing

The Earley parsing method is inherently left to right. Instead of merely a non-terminal symbol, as with CYK items, an Earley item holds an entire production. It uses a dot in the right-hand side of the production to indicate up to which point a parse has completed. The part left of the dot represents the part that is already parsed, and the part right of the dot represents the part that still needs to be done. A dot at the end of a production indicates that the entire production has been recognized. A dot can be moved past a symbol as soon as an item that indicates the completed recognition of that symbol is available. This is captured by the scan and complete deduction rules. The init and predict deductions introduce possible candidates for parsing. The schema is as follows.

**Parsing Schema 3.4 (E).**

$$\begin{aligned}
\mathcal{I}_E &= \{[A \rightarrow \alpha \bullet \beta, i, j] \mid A \rightarrow \alpha \beta \in P \wedge 0 \leq i \leq j\} \\
\mathcal{H}_E &= \{[a, i - 1, i] \mid a = a_i \wedge 1 \leq i \leq n\} \\
\mathcal{D}^{\text{Init}} &= \{\vdash [S \rightarrow \bullet \gamma, 0, 0]\} \\
\mathcal{D}^{\text{Predict}} &= \{[A \rightarrow \alpha \bullet B \beta, i, j] \vdash [B \rightarrow \bullet \gamma, j, j]\} \\
\mathcal{D}^{\text{Scan}} &= \{[A \rightarrow \alpha \bullet a \beta, i, j], [a, j, j + 1] \vdash [A \rightarrow \alpha a \bullet \beta, i, j + 1]\} \\
\mathcal{D}^{\text{Complete}} &= \{[A \rightarrow \alpha \bullet B \beta, i, j], [B \rightarrow \gamma \bullet, j, k] \vdash [A \rightarrow \alpha B \bullet \beta, i, k]\} \\
\mathcal{D}_E &= \mathcal{D}^{\text{Init}} \cup \mathcal{D}^{\text{Predict}} \cup \mathcal{D}^{\text{Scan}} \cup \mathcal{D}^{\text{Complete}}
\end{aligned}$$

This parser yields the following set of recognized items:

$$\{[A \rightarrow \alpha \bullet, i, j] \mid \alpha \Rightarrow^* a_i \dots a_j \wedge S \Rightarrow^* a_1 \dots a_i A \gamma \text{ for some } \gamma\}.$$

The parser recognizes the input string if  $[S \rightarrow \alpha \bullet, 0, n]$ , where  $\alpha \Rightarrow^* a_1 \dots a_n$ .

The Earley schema, as presented, is inherently a top-down parser (top-down filtering, see  $\mathcal{D}^{\text{Init}}$  and  $\mathcal{D}^{\text{Predict}}$ ). As we discussed, this reduces the possibilities for parallelization. We can obtain a purely bottom-up variant of **E**, by altering the deduction steps of **E**. The replacement steps for the bottom-up Earley variant, short **buE**, are given in the following schema.

**Parsing Schema 3.5 (buE).**

$$\begin{aligned}
\mathcal{D}^{\text{Init}} &= \{\vdash [A \rightarrow \bullet \gamma, i, i]\} \\
\mathcal{D}^{\text{Scan}} &= \{[A \rightarrow \alpha \bullet a \beta, i, j], [a, j, j + 1] \vdash [A \rightarrow \alpha a \bullet \beta, i, j + 1]\} \\
\mathcal{D}^{\text{Complete}} &= \{[A \rightarrow \alpha \bullet B \beta, i, j], [B \rightarrow \gamma \bullet, j, k] \vdash [A \rightarrow \alpha B \bullet \beta, i, k]\} \\
\mathcal{D}_{\text{buE}} &= \mathcal{D}^{\text{Init}} \cup \mathcal{D}^{\text{Scan}} \cup \mathcal{D}^{\text{Complete}}
\end{aligned}$$

This parser yields the following set of recognized items:

$$\{[A \rightarrow \alpha \bullet, i, j] \mid \alpha \Rightarrow^* a_i \dots a_j\}.$$

The set of correct final items is identical to the set associated with  $\mathbf{E}$ . Obviously, since the  $D^{\text{Init}}$  of  $\mathbf{buE}$  produces more items than the  $D^{\text{Init}}$  and  $D^{\text{Predict}}$  of  $\mathbf{E}$  do together, the set of items  $\mathbf{buE}$  recognizes is larger than with  $\mathbf{E}$ . This variant, however, is better suited for parallel processing. The canonical Earley method can be obtained from bottom-up Earley by applying a static filter, i.e.,  $\mathbf{buE} \xrightarrow{\text{sf}} \mathbf{E}$ .

There are several other optimizations that we can apply to  $\mathbf{E}$ . First, we consider the possibility of incorporating a look-ahead. For this, we need to include a *end-of-string marker*  $\$$  in our grammar. Given an arbitrary context-free grammar  $G \in \mathcal{CFG}$ , for which  $\{S', \$\} \cap V = \emptyset$ , we define a new grammar  $G'$ :

$$\begin{aligned} N' &= N \cup \{S'\}, \\ \Sigma' &= \Sigma \cup \{\$\}, \\ P' &= P \cup \{S' \rightarrow S\$\}, \\ G' &= \langle N', \Sigma', P', S' \rangle. \end{aligned}$$

We now only have one final item, viz.  $[S' \rightarrow S \bullet \$, 0, n]$ . Furthermore, we define the following function  $\text{FOLLOW} : N \rightarrow \wp(\Sigma')$  by

$$\text{FOLLOW}(A) = \{a \mid \exists \alpha, \beta : S' \Rightarrow^* \alpha A a \beta\}$$

We can now define the following schema.

**Parsing Schema 3.6 (E(1)).**

$$\begin{aligned} \mathcal{I}_{\mathbf{E}(1)} &= \{[A \rightarrow \alpha \bullet \beta, i, j] \mid A \rightarrow \alpha \beta \in P' \wedge 0 \leq i \leq j\} \\ \mathbf{H}_{\mathbf{E}(1)} &= \{[a, i-1, i] \mid a = a_i \wedge 1 \leq i \leq n\} \cup \{[\$, n, n+1]\} \\ \mathbf{D}^{\text{Init}} &= \{\vdash [S' \rightarrow \bullet S \$, 0, 0]\} \\ \mathbf{D}^{\text{Predict}} &= \{[A \rightarrow \alpha \bullet B \beta, i, j] \vdash [B \rightarrow \bullet \gamma, j, j]\} \\ \mathbf{D}^{\text{Scan}} &= \{[A \rightarrow \alpha \bullet a \beta, i, j], [a, j, j+1] \vdash [A \rightarrow \alpha a \bullet \beta, i, j+1]\} \\ \mathbf{D}^{\text{Complete}} &= \{[A \rightarrow \alpha \bullet B \beta, h, i], [B \rightarrow \gamma \bullet, i, j], [a, j, j+1] \\ &\quad \vdash [A \rightarrow \alpha B \bullet \beta, h, j] \mid a \in \text{FOLLOW}(B)\} \\ \mathbf{D}_{\mathbf{E}(1)} &= \mathbf{D}^{\text{Init}} \cup \mathbf{D}^{\text{Predict}} \cup \mathbf{D}^{\text{Scan}} \cup \mathbf{D}^{\text{Complete}} \end{aligned}$$

More sophisticated look-ahead filtering is possible, for example by using  $a \in \text{FIRST}(\beta \text{FOLLOW}(A))$  or applying a similar filter to  $\mathbf{D}^{\text{scan}}$ . This, however, is exactly the type of filtering the SLR(1) parser uses. It can be shown that an SLR(1) parser is an implementation

of **E**. However, an SLR(1) parser is, like an LR parser, defined for a subclass of  $\mathcal{CFG}$ , viz. deterministic context-free grammars. Finally,  $\mathbf{E} \xRightarrow{\text{df}} \mathbf{E(1)}$  holds.

Another optimization, by means of step contraction, is given by Graham, Harrison, and Ruzzo. The parsing method comprises two optimizations:

- *nullable symbols* are skipped when the dot is worked rightwards through a production;
- derivations of the form  $A \Rightarrow^+ B$  (*chain derivations*) are reduced to single steps.

Its parsing schema is given in [SN97].

#### Parsing Schema 3.7 (GHR).

$$\begin{aligned}
\mathcal{I}_{\text{GHR}} &= \{[A \rightarrow \alpha \bullet \beta, i, j] \mid A \rightarrow \alpha \beta \in P \wedge 0 \leq i \leq j\} \\
D^{\text{Init}} &= \{\vdash [S \rightarrow \bullet \gamma, 0, 0] \mid \beta \Rightarrow^* \epsilon\} \\
D^{\text{Scan}} &= \{[A \rightarrow \alpha \bullet a \beta \gamma, i, j], [a, j, j + 1] \\
&\quad \vdash [A \rightarrow \alpha a \beta \bullet \gamma, i, j + 1] \mid \beta \Rightarrow^* \epsilon\} \\
D^{C1} &= \{[A \rightarrow \alpha \bullet B \beta \gamma, i, j], [B \rightarrow \delta \bullet, j, k] \\
&\quad \vdash [A \rightarrow \alpha B \beta \bullet \gamma, i, k] \mid i < j < k \wedge \beta \Rightarrow^* \epsilon\} \\
D^{C2} &= \{[A \rightarrow \alpha \bullet B \beta \gamma, i, j], [C \rightarrow \delta \bullet, i, j] \\
&\quad \vdash [A \rightarrow \alpha B \beta \bullet \gamma, i, j] \mid i < j \wedge \beta \Rightarrow^* C \wedge \beta \Rightarrow^* \epsilon\} \\
D^{\text{Predict}} &= \{[A \rightarrow \alpha \bullet B \beta, i, j] \vdash [C \rightarrow \alpha' \bullet \beta', j, j] \mid B \Rightarrow^* C \wedge \alpha' \Rightarrow^* \epsilon\} \\
D_{\text{E}} &= D^{\text{Init}} \cup D^{\text{Scan}} \cup D^{C1} \cup D^{C2} \cup D^{\text{Predict}}
\end{aligned}$$

It can be proven that  $\mathbf{E} \xRightarrow{\text{sc}} \mathbf{GHR}$  holds.

### 3.5 Double Dotted Parsing

The Double Dotted Parser was originally introduced by [dVH89, dVH91]. Its main characteristic is that it allows parsing not only from left to right, but also from right to left. Parsing can start at any point in the input string. This makes the parsing method well-suited for parallel processing.

#### Parsing Schema 3.8 (DD).

$$\begin{aligned}
\mathcal{I}_{\text{DD}} &= \{[A \rightarrow \alpha \bullet \beta \bullet \gamma, i, j] \mid A \rightarrow \alpha \beta \gamma \in P \\
&\quad \wedge 0 \leq i \leq j \wedge (\beta \neq \epsilon \text{ or } \alpha \gamma = \epsilon)\} \\
D^{\text{Init}} &= \{[a, j - 1, j] \vdash [A \rightarrow \alpha \bullet a \bullet \gamma, j - 1, j]\} \\
D^{\epsilon} &= \{\vdash [B \rightarrow \bullet \bullet, j, j]\} \\
D^{\text{Include}} &= \{[B \rightarrow \bullet \beta \bullet, i, j] \vdash [A \rightarrow \alpha \bullet B \bullet \gamma, i, j]\}
\end{aligned}$$

$$\begin{aligned}
D^{\text{Concatenate}} &= \{[A \rightarrow \alpha \bullet \beta_1 \bullet \beta_2 \gamma, i, j], [A \rightarrow \alpha \beta_1 \bullet \beta_2 \bullet \gamma, j, k] \\
&\quad \vdash [A \rightarrow \alpha \bullet \beta_1 \beta_2 \bullet \gamma, i, k]\} \\
D^{\text{DD}} &= D^{\text{Init}} \cup D^\epsilon \cup D^{\text{Include}} \cup D^{\text{Concatenate}}
\end{aligned}$$

A valid item  $[A \rightarrow \alpha \bullet \beta \bullet \gamma]$  represents the fact that  $\beta \Rightarrow^* a_{i+1} \dots a_j$  has been recognized.

An algorithm implementing this parsing method will perform many redundant operations. In the worst case, a valid item of the form  $[A \rightarrow \alpha \bullet \beta \bullet \gamma]$ , with  $|\beta| = m$ , can be the result of an concatenation of two items  $m - 1$  times. Similarly, each of these smaller items may have been obtained in various ways as well. To overcome this redundancy, we can limit the concatenate step to allow only a single symbol to be concatenated to the right. This is sufficient to parse any string. The replacement concatenate step for the enhanced double dotted parser is shown in schema 3.9.

#### Parsing Schema 3.9 ( $\text{DD}^{\text{lr}}$ ).

$$\begin{aligned}
D^{\text{Concatenate}} &= \{[A \rightarrow \alpha \bullet \beta \bullet X \gamma, i, j], [A \rightarrow \alpha \beta \bullet X \bullet \gamma, j, k] \\
&\quad \vdash [A \rightarrow \alpha \bullet \beta X \bullet \gamma, i, k]\}
\end{aligned}$$

After further inspection, it becomes clear that many of the items in the domain of  $\text{DD}^{\text{lr}}$  can never be successfully used to obtain new valid items (viz.  $\{[A \rightarrow \alpha \bullet \beta \bullet \gamma, i, j] \mid |\alpha| \geq 1 \wedge |\beta| \geq 2\}$ ). By removing these items from the domain, and adjusting the deduction steps accordingly, we obtain the following schema (for more details, see [SN97]).

#### Parsing Schema 3.10 ( $\text{DD}^{\text{lro}}$ ).

$$\begin{aligned}
\mathcal{I}^{(1)} &= \{[A \rightarrow \alpha \bullet X \bullet \gamma, i, j] \mid A \rightarrow \alpha X \gamma \in P \wedge 0 \leq i \leq j\} \\
\mathcal{I}^{(2)} &= \{[A \rightarrow X \beta \bullet \gamma, i, j] \mid A \rightarrow X \beta \gamma \in P \wedge 0 \leq i \leq j\} \\
\mathcal{I}^{(3)} &= \{[A \rightarrow \bullet \bullet, j, j] \mid A \rightarrow \epsilon P \wedge j \geq 0\} \\
\mathcal{I}_{\text{DD}^{\text{lro}}} &= \mathcal{I}^{(1)} \cup \mathcal{I}^{(2)} \cup \mathcal{I}^{(3)} \\
D^{\text{Init}} &= \{[a, j - 1, j] \vdash [A \rightarrow \alpha \bullet a \bullet \gamma, j - 1, j]\} \\
D^\epsilon &= \{\vdash [B \rightarrow \bullet \bullet, j, j]\} \\
D^{\text{Include}} &= \{[B \rightarrow \bullet \beta \bullet, i, j] \vdash [A \rightarrow \alpha \bullet B \bullet \gamma, i, j]\} \\
D^{\text{Concatenate}} &= \{[A \rightarrow \bullet \alpha \bullet X \gamma, i, j], [A \rightarrow \alpha \bullet X \bullet \gamma, j, k] \vdash [A \rightarrow \bullet \alpha X \bullet \gamma, i, k]\} \\
D_{\text{DD}^{\text{lro}}} &= D^{\text{Init}} \cup D^\epsilon \cup D^{\text{Include}} \cup D^{\text{Concatenate}}
\end{aligned}$$

All these optimizations have been static filters, so  $\text{DD} \xRightarrow{\text{sf}} \text{DD}^{\text{lr}} \xRightarrow{\text{sf}} \text{DD}^{\text{lro}}$  holds. Each of these algorithms can be enhanced with look-back and look-ahead (dynamic filters). An implementation of  $\text{DD}^{\text{lr}}$ , augmented with look-back and look-ahead, can be found in [dVH89].

### 3.6 Left- and Head-corner Parsing

Here we will describe the underlying parsing method for the generalized left-corner parsing algorithm (e.g. [MTH<sup>+</sup>83, Ned93]), as opposed to deterministic left-corner parsing (see [RL70]). Generalized LC parsing is a step contraction of canonical Earley:  $\mathbf{E} \xrightarrow{\text{sc}} \mathbf{LC}$ . As a result the domain (item set) shrinks considerably. It is straightforward to transform  $\mathbf{buE}$  to a bottom-up left-corner schema  $\mathbf{buLC}$ , so we will first consider  $\mathbf{buLC}$ , instead of the more elaborate  $\mathbf{LC}$ .

First, we consider the basic idea of left-corner parsing. Consider an item  $[A \rightarrow B\bullet\beta, i, j]$  in  $\mathbb{P}_{\text{buE}}$ . This item can only be valid if we already had the valid items  $[B \rightarrow \gamma\bullet, i, j]$  and  $[A \rightarrow \bullet B\beta, i, i]$ , of which the latter always valid by  $D_{\text{Init}}$ . So we can replace the steps

$$\begin{aligned} & \vdash [A \rightarrow \bullet B\beta, i, i] \\ & [A \rightarrow \bullet B\beta, i, i], [B \rightarrow \gamma\bullet, i, j] \vdash [A \rightarrow B\bullet\beta, i, j] \end{aligned}$$

by

$$[B \rightarrow \gamma\bullet, i, j] \vdash [A \rightarrow B\bullet\beta, i, j].$$

A similar step contraction can be applied to the scan step. The resulting parsing schema is as follows.

**Parsing Schema 3.11 (buLC).**

$$\begin{aligned} \mathcal{I}^{(1)} &= \{[A \rightarrow X\alpha\bullet\beta, i, j] \mid A \rightarrow X\alpha\beta \in P \wedge 0 \leq i \leq j\} \\ \mathcal{I}^{(2)} &= \{[A \rightarrow \bullet, j, j] \mid A \rightarrow \epsilon \in P \wedge j \geq 0\} \\ \mathcal{I}_{\text{buLC}} &= \mathcal{I}^{(1)} \cup \mathcal{I}^{(2)} \\ D^\epsilon &= \{ \vdash [A \rightarrow \bullet, j, j] \} \\ D^{\text{LC}(a)} &= \{[a, j-1, j] \vdash [B \rightarrow a\bullet\beta, j-1, j]\} \\ D^{\text{LC}(A)} &= \{[A \rightarrow \alpha\bullet, i, j] \vdash [B \rightarrow A\bullet\beta, i, j]\} \\ D^{\text{Scan}} &= \{[A \rightarrow \alpha\bullet a\beta, i, j], [a, j, j+1] \vdash [A \rightarrow \alpha a\bullet\beta, i, j+1]\} \\ D^{\text{Complete}} &= \{[A \rightarrow \alpha\bullet B\beta, i, j], [B \rightarrow \gamma\bullet, j, k] \vdash [A \rightarrow \alpha B\bullet\beta, i, k]\} \\ D_{\text{buLC}} &= D^\epsilon \cup D^{\text{LC}(a)} \cup D^{\text{LC}(A)} \cup D^{\text{Scan}} \cup D^{\text{Complete}} \end{aligned}$$

Note that the number of steps that is performed by the complete and scan deductions is reduced, because the item domain is limited. The item set can only contain items of which the dot indicates that (at least) the first symbol of the production has been parsed (except for empty productions). Scanning the first terminal for some production is now done by  $D^{\text{LC}(a)}$ , instead of  $D^{\text{Scan}}$ .

The regular generalized LC parsing is much more complex, because in the case of  $\mathbf{E}$ ,  $[A \rightarrow \bullet B\beta, i, i]$  is not always valid. Instead of assuming its validity, we need to deduce its validity. For this, looking at Earley's prediction rule, there need be some valid item

of the form  $[C \rightarrow \alpha \bullet A \delta, h, i]$ . This poses no problem if  $\alpha \neq \epsilon$ , but if  $\alpha = \epsilon$ , we need to seek further for items that validate  $[C \rightarrow \bullet A \delta, i, i]$  (note the recursion). In order to define the correct solution to this problem, we need to define the left corner relation.

**Definition 3.15 (left-corner relation).** The *left corner* of a non-empty production is the left most symbol on the right-hand side of a production; The left corner of an empty production is  $\epsilon$ . We define  $>_l: N \times (N \cup \Sigma \cup \{\epsilon\})$  as

$$>_l = \{(A, U) \mid A \rightarrow U \alpha \in P\} \cup \{(A, \epsilon) \mid A \rightarrow \epsilon \in P\}$$

The transitive and reflexive closure of  $>_l$  is denoted  $>_l^*$ .

We can now give the solution.

**Parsing Schema 3.12 (LC).**

$$\begin{aligned} \mathcal{I}^{(1)} &= \{[A \rightarrow X \alpha \bullet \beta, i, j] \mid A \rightarrow X \alpha \beta \in P \wedge 0 \leq i \leq j\} \\ \mathcal{I}^{(2)} &= \{[A \rightarrow \bullet, j, j] \mid A \rightarrow \epsilon \in P \wedge j \geq 0\} \\ \mathcal{I}^{(3)} &= \{[S \rightarrow \bullet \gamma, 0, 0] \mid S \rightarrow \gamma \in P\} \\ \mathcal{I}_{LC} &= \mathcal{I}^{(1)} \cup \mathcal{I}^{(2)} \cup \mathcal{I}^{(3)} \\ D^{Init} &= \{\vdash [S \rightarrow \bullet \gamma, 0, 0]\} \\ D^{LC(\epsilon)} &= \{[C \rightarrow \gamma \bullet E \delta, h, i] \vdash [B \rightarrow \bullet, i, i] \mid E >_l^* B\} \\ D^{LC(a)} &= \{[C \rightarrow \gamma \bullet E \delta, h, i], [a, i, i + 1] \\ &\quad \vdash [B \rightarrow a \bullet \beta, i, i + 1] \mid E >_l^* B\} \\ D^{LC(A)} &= \{[C \rightarrow \gamma \bullet E \delta, h, i], [A \rightarrow \alpha \bullet, i, j] \\ &\quad \vdash [B \rightarrow A \bullet \beta, i, j] \mid E >_l^* B\} \\ D^{Scan} &= \{[A \rightarrow \alpha \bullet a \beta, i, j], [a, j, j + 1] \vdash [A \rightarrow \alpha a \bullet \beta, i, j + 1]\} \\ D^{Complete} &= \{[A \rightarrow \alpha \bullet B \beta, i, j], [B \rightarrow \gamma \bullet, j, k] \vdash [A \rightarrow \alpha B \bullet \beta, i, k]\} \\ D_{LC} &= D^{LC(\epsilon)} \cup D^{LC(a)} \cup D^{LC(A)} \cup D^{Scan} \cup D^{Complete} \end{aligned}$$

As with their corresponding Earley parsers, **buLC**  $\xRightarrow{\text{df}}$  **LC** holds.

The left-corner parsing methods relate in an interesting way with double dotted parsing methods, namely **DD**<sup>lro</sup>  $\xRightarrow{\text{sc}}$  **buLC** and **DD**  $\xRightarrow{\text{sc}}$  **LC**. In general, however, this does not imply that left-corner parsers are better than double dotted parsers. For example, **DD** can parse both left to right and right to left, which makes it more flexible.

A slight modification to the filter that transform an double dotted parser into a left-corner parser can yield a head-corner parser. A head-corner parser is based on the heuristic that it is best to start parsing at some most relevant word. In other words, instead of considering the left most symbol of the right-hand side of an production the most important, an arbitrary symbol on the right-hand side is considered most important. The idea is that introducing items only when there exists a partial parse for the head symbol of the production improves the performance.

Head-corner parsers need a slight addition to context-free grammar, viz. to indicate the head of a production rule. A context-free head grammar is a 5-tuple  $\langle N, \Sigma, P, S, r \rangle$ , where  $N, \Sigma, P$ , and  $S$  are as usual, and  $r : P \rightarrow \mathbb{N}$  is a function returning a natural number for some  $p \in P$ , indicating which symbol on the right-hand of  $p$  should be considered the head. We write  $|p|$  for the right-hand of  $p$ . If  $|p| = \epsilon$ ,  $r(p)$  must be 0, else  $1 \leq r(p) \leq |p|$ .

Similar to  $>_l$ , we define a head corner relation as follows:

**Definition 3.16 (head-corner relation).** The *head-corner* of a non-empty production  $p$  is the  $r(p)^{\text{th}}$  symbol on the right-hand side of that production. We denote a head by underlining it. The head-corner of an empty production is  $\epsilon$ . We define  $>_h : N \times (N \cup \Sigma \cup \{\epsilon\})$  as

$$>_h = \{(A, U) \mid A \rightarrow \alpha \underline{U} \beta \in P\} \cup \{(A, \epsilon) \mid A \rightarrow \epsilon \in P\}$$

The transitive and reflexive closure of  $>_h$  is denoted  $>_h^*$ .

Since we start parsing a production from an arbitrary point in its right-hand side, we need to use double dotted items. Parsing may proceed in both directions. To reduce the already large number of deduction steps that need to be specified we introduce a *predict item*. We could have introduced this item with **LC** as well, by replacing  $[C \rightarrow \gamma \cdot E \delta, h, i]$  in the deduction steps  $D^{LC(\epsilon)}$ ,  $D^{LC(a)}$  and  $D^{LC(A)}$  with  $[i, E]$  and introducing the deduction step  $D^{\text{Predict}}$ :

$$[C \rightarrow \gamma \cdot E \delta, h, i] \vdash [i, E].$$

We obtain the following schema, which is a slight modification of the schema in [SodA96].

**Parsing Schema 3.13 (HC).**

$$\begin{aligned} \mathcal{I}^{\text{Predict}} &= \{[i, j, A] \mid A \in N \wedge 0 \leq i \leq j\} \\ \mathcal{I}^{(1)} &= \{[A \rightarrow \alpha \bullet \beta X \gamma \bullet \delta, i, j] \mid A \rightarrow \alpha \bullet \beta \underline{X} \gamma \bullet \delta \in P \wedge 0 \leq i \leq j\} \\ \mathcal{I}^{(2)} &= \{[A \rightarrow \bullet \bullet, j, j] \mid A \rightarrow \epsilon \in P \wedge j \geq 0\} \\ \mathcal{I}_{\text{HC}} &= \mathcal{I}^{\text{Predict}} \cup \mathcal{I}^{(1)} \cup \mathcal{I}^{(2)} \\ D^{\text{Init}} &= \{\vdash [0, n, S]\} \\ D^{\text{HC}(\epsilon)} &= \{[l, r, A] \vdash [B \rightarrow \bullet \bullet, i, i] \mid A >_l^* B \wedge l \leq j \leq r\} \\ D^{\text{HC}(a)} &= \{[l, r, A], [a, i, i + 1] \\ &\quad \vdash [B \rightarrow \alpha \bullet a \bullet \beta, i, i + 1] \mid A >_l^* B \wedge l < j \leq r\} \\ D^{\text{HC}(A)} &= \{[l, r, A], [C \rightarrow \bullet \gamma \bullet, i, j] \\ &\quad \vdash [B \rightarrow \alpha \bullet C \bullet \beta, i, j] \mid A >_l^* B \wedge l \leq i \leq j \leq r\} \\ D^{\text{Predict}^1} &= \{[l, r, A], [B \rightarrow \alpha C \bullet \beta \bullet \gamma] \\ &\quad \vdash [l, i, C] \mid A >_h^* B \wedge l \leq i \leq j \leq r\} \\ D^{\text{Predict}^2} &= \{[l, r, A], [B \rightarrow \alpha C \bullet \beta \bullet \gamma] \end{aligned}$$



$$\begin{aligned}
& \vdash [l, i, C] \mid A >_h^* B \wedge l \leq i \leq j \leq r \} \\
D^{\text{Scan}^1} &= \{ [l, r, A], [a, j-1, j], [B \rightarrow \alpha a \bullet \beta \bullet \gamma, j, k] \\
& \quad \vdash [B \rightarrow \alpha \bullet a \beta \bullet \gamma, j-1, k] \mid A >_h^* B \wedge l < j \leq k \leq r \} \\
D^{\text{Scan}^2} &= \{ [l, r, A], [B \rightarrow \alpha \bullet \beta \bullet a \gamma, i, j-1], [a, j-1, j] \\
& \quad \vdash [B \rightarrow \alpha \bullet \beta a \bullet \gamma, i, j] \mid A >_h^* B \wedge l < j \leq k \leq r \} \\
D^{\text{Complete}^1} &= \{ [l, r, A], [C \rightarrow \bullet \delta \bullet, i, j], [B \rightarrow \alpha C \bullet \beta \bullet \gamma, j, k] \\
& \quad \vdash [B \rightarrow \alpha \bullet C \beta \bullet \gamma, i, k] \mid A >_h^* B \wedge l \leq i \leq j \leq k \leq r \} \\
D^{\text{Complete}^2} &= \{ [l, r, A], [B \rightarrow \alpha \bullet \beta \bullet C \gamma, i, j], [C \rightarrow \bullet \delta \bullet, j, k] \\
& \quad \vdash [B \rightarrow \alpha \bullet \beta C \bullet \gamma, i, k] \mid A >_h^* B \wedge l \leq i \leq j \leq k \leq r \} \\
D_{\text{HC}} &= D^{\text{HC}(\epsilon)} \cup D^{\text{HC}(a)} \cup D^{\text{HC}(A)} \cup D^{\text{Predict}^1} \cup D^{\text{Predict}^2} \\
& \quad \cup D^{\text{Scan}^1} \cup D^{\text{Scan}^2} \cup D^{\text{Complete}^1} \cup D^{\text{Complete}^2}
\end{aligned}$$

Head-corner parsing seems especially useful if the context-free grammar is augmented with feature structures.



## Chapter 4

# Parallel Parsing Strategies

This chapter contains an overview of different approaches to the implementation of parallel algorithms for parsing methods. We will give a brief outline of the underlying idea of each approach, thereby only focusing on the essentials. If the approach is based on an originally sequential algorithm, we will give a brief outline of the sequential parsing algorithm, after which we will proceed with the possible parallelization techniques. After each outline, we will discuss for what specific architectures the approach is (known to be) suitable, and how the approach relates to any of the other methods.

A fairly complete overview of parallel parsing techniques is given in [Nij91] and [AH94]. The overview in this chapter, has largely been taken from these papers, occasionally augmented with more recent advances and some additional experimental data.

Originally, parallel parsing was explored by continuing with the traditional LL-, LR-, and precedence parsing. Parallel parsing was conducted by using multiple serial parsers in parallel. With the rise of massively parallel architectures, it became more interesting to investigate other parsing techniques. Most notably, CYK- and Early-style parallel parsers can parse in  $O(n)$  time. Other, more theoretical, research tried to find lower bounds for the time and space complexity, often by designing algorithms for a subclass of context-free languages. [GR88] gives an introduction to this field. A typical result from this field is that context-free languages can be parsed in  $O(\log_2 n)$  time using  $n^6$  processors, for a given input string with length  $n$ . For practical implementations, however,  $O(n^6)$  processors is not a reasonable requirement.

### 4.1 From One to Many Traditional Parsers

The most obvious approach to parallelization of a sequential parser is to use several sequential parsers on different parts of the problem in parallel. For example, a possible setup for 2 processors is to have one parser processing input from the left, and another processing input from the right. When the two meet, their partial parses have to be combined into a single parse tree. This idea can be generalized to having many parsers operating on different parts of an input string.

#### 4.1.1 Parsing Separate Parts of the Input String

There are several solutions for parallel parsing that are based on this approach. [Fis75] gives an example of an LR parser that follows this strategy. Fisher introduces synchronous parsing machines (SPMs), which can LR-parse a part of the input string. Each SPM is a slightly modified serial parser. In theory, SPMs can start at any point of the input string, but in practical situations, it is best to start parsing at, for example, the beginning of a statement or block (in case of a programming language).

Obviously, SPMs cannot start parsing in the regular initial state. Instead, each SPM starts parsing with a set of states, guaranteed to contain the correct one. For each state, a separate stack is maintained. The set of ‘initial’ states for an SPM can be determined by choosing the states that are consistent with the first symbol that is about to be parsed.

When an SPM comes to the point where its right-hand neighbor SPM started, an attempt will be made to merge the partial results of the respective SPMs. Obviously, if the left-hand SPM uses only one stack, it is possible to determine what the combined result of the two SPMs should become, i.e., which of the stacks of the right-hand SPM yields the correct parse. If the left-hand SPM uses more than one stack, the merge will be delayed until the SPM to its left reaches the respective SPM, using only one stack. Since the left-most SPM (which starts at position 1), always uses one state, viz. the regular initial state, there will eventually always be a left-hand SPM that uses one stack. For more information on this parser see [Nij91].

#### 4.1.2 Parallel Processing in case of Non-Determinism

One problem with LR parsers is that the LR-table will contain conflicting entries in case of a non-LR-grammar. Tomita’s solution to this problem ([Tom91]) can be seen as using multiple LR-parsers in Parallel (in case of non-determinism). In Tomita’s algorithm, each time the parser encounters a conflicting entry in the parse table, the current process is copied for each additional entry. To avoid copying the stack each time a new process is spawned, the algorithm uses a so called “graph-structured” stack, which is a single data-structure representing the stacks of all processes. The processes are synchronized on the shift action.

Tomita’s work has mainly been concerned with the creation of an efficient sequential implementation for full context-free parsing. As is argued in [Nij91], Tomita’s solution is not usable as a solution specifically designed for parallel machines. The graph-structured stack requires a master process. Moreover, each input word need to be supplied to each process, which is inefficient.

If we let go of the idea of a graph-structured stack, we come back to the idea of letting the spawned processes operate independently on their own copy of the stack. In this case, each spawned process needs to have a full copy of both the stack and the part of the input string which still needs to be parsed. The communication costs are obviously too high for practical implementations, especially if the amount of non-determinism is large.

Finally, [Nij94] suspects serious problems if Tomita’s algorithm is used for on-line parsing.

When other approaches for parallelizing Tomita’s algorithm are pursued, Tomita’s

method becomes closely related to Earley's method, which is covered in the next section. A cross-breeding of Tomita's method and Earley style parsing is given in [Sik93b].

## 4.2 Parallel Chart Parsing

The essential differences between the basic chart, string chart, and tabular chart versions are the data structures they use. In increasing order, they use a refined data structure resulting in an algorithm that requires fewer operations to complete. From a serial point of view, the tabular chart parsers seem to perform best. [Nij94] investigates parallelization techniques for all of the mentioned chart parsers. As we will see, the straightforwardly parallelized version of these parsers all seem to be suited for a different specific architecture.

Although initialization and the specific rules are different for each parser type (CYK, Earley, or Double Dotted; see chapter 3), the concept of the parsing approach remains the same. The key concern is to gain insight in how each approach structures the data, and the possibilities for parallelization. For a more detailed description of the parsers refer to [Nij94].

### 4.2.1 Basic Chart Parsing

The basic chart approach uses two lists: an agenda and a working area. The agenda contains items that need processing, whereas the working area contains the items being processed. After initialization, parsing continues by removing an (arbitrary) entry from the agenda, and putting it on the work area. It is compared against other items in the work area, and depending on some rules specific to the underlying parser (scan, predict, and complete for Earley; inclusion and concatenation for Double Dotted), new items are added to the agenda. The parse succeeds if the item  $[0, S, n]$  can be found in the work area.

The algorithm gives a lot of freedom with respect to the order of evaluation. On the other hand, each time an item is selected, all items in the working area have to be traversed, which is rather inefficient.

#### Parallelization Techniques

We mentioned that the choice of which item to remove from the agenda was arbitrary. We can imagine multiple processors taking items of the agenda, into the work area, and starting processing them. A sketch of this approach is given by figure 4.1.

Once the item is off the agenda—and in the work area—the process that is handling the item need only refer to the items on the work area and to the original production rules.

The parallelism can even be made more fine grained, by assigning multiple processors to each item being processed, provided that the execution of different (parts of the) rules that are used by a specific parser can be parallelized (e.g., 2 processors can work in parallel on the CYK rule).

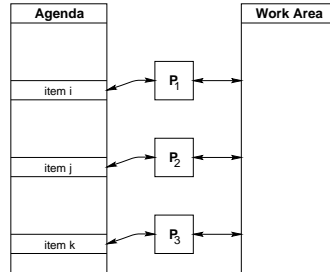


Figure 4.1: Three processors working on a basic parse chart.

### Parallel Architectures

[Nij94] notes that the amount of processors that can effectively be used with the basic chart approach is limited. It is mainly suitable for limited processor shared-memory MIMD architectures. An implementation of this approach for an Earley parser has been given in [GC88].

These findings are backed up by the experiments conducted by [Tho91]. Most of the experiments given in this section considered a shared-memory MIMD machine. [Tho91] considers parallel chart parsers for loosely coupled parallel systems. More interestingly, he conducted his experiments on real (non-simulated) parallel architectures. Contrary to most of the theoretical research, this research focused on minimizing the communication cost.

It is mentioned that (the maximum) linear speedup that can be achieved with loosely coupled systems is most likely to be obtained for problems that are isomorphic to a tree-search problem, where the initial fan-out is large, and the specifications of subproblems can be compact. In these cases, the distribution of sub-problems requires the representation of partial solutions. Based on this knowledge, [Tho91] considers active chart parsers as the most likely candidates for efficient parallel parsers (this is not the common belief though).

A straightforward parallel implementation of the active chart on a shared memory machine (the BBN Butterfly<sup>TM</sup>) achieved the expected linear speedup, for a limited number of processors. Later research ([Tho94]) showed no improvement for larger numbers (more than 3) of processors. Experiments on the Intel Hypercube<sup>TM</sup> and Xerox 1186 showed that the communication costs that resulted from edge distribution heavily out-weighed the computation costs. So no speedup was achieved.

#### 4.2.2 String Chart Parsing

The string chart approach refines the data structure of the basic chart approach by assigning an item set  $I_i$  to each  $a_i$  of the input string (in the case of Earley also  $I_0$ ). After initializing the left most item-set, the item sets are constructed in a left to right manner. Each further set is initialized similar to the first one. A new item is added if an item in some set matches (is adjacent to) an item in any of the previous sets. Its position marker can be used to determine which of the previously constructed sets

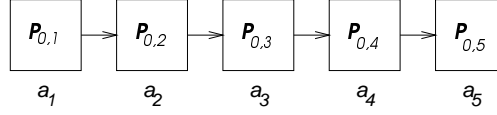


Figure 4.2: Three processors working on a basic parse chart.

should be examined.

All items that are deduced in a specific item set belong to a predetermined subclass of the item domain of underlying parsing method. More specifically, the set of items  $I_i$  that can be deduced for an item set conforms to a subset  $\mathcal{I}_i$  of the domain  $\mathcal{I}$  of the parsing method that is implemented with the string chart parser. The limited domain  $\mathcal{I}_k$  of an item set  $I_k$ , where  $k = 1, \dots, n$ , is as follows:

$$\mathcal{I}_k = \{[\phi, i, k] \in \mathcal{I}\}, \text{ for each } k = 1, \dots, n, \quad (4.1)$$

where  $\phi$  can be anything, depending on the parsing method. Furthermore,  $\bigcap_{i=1}^n \mathcal{I}_i = \emptyset$  holds. Depending on the specific implementation, item sets may actually contain more items, but only if they were generated for another item set. We will call the items in an item set  $I_k$  that are an element of the corresponding  $\mathcal{I}_k$  the primary items of  $I_k$ . The parser accepts the input string if an item representing  $S \Rightarrow^* a_1 \dots a_n$  is contained in set  $I_n$ .

### Parallelization Techniques

It is possible to assign a processor to each item set, or  $a_i$ , as is shown in figure 4.2. By distinguishing the same set of items as with the tabular chart version (see below), we can apply the following scheme. First, each processor computes the set of items that would appear in cell  $t_{j-1,j}$  of a parse matrix, as soon as the required data becomes available. After that, the processor can proceed computing the items that would appear in  $t_{j-2,j}$ . This scheme yields a correct computation order, while computing the parse in a pipelined fashion. Each processor needs  $O(j^2)$  time to complete its computations. So, this approach results in algorithms that need  $O(n^2)$  time and require  $O(n)$  memory per processor. In general, the string chart parser can be viewed upon as a sequence of basic chart parsers, because each processor maintains its own internal agenda.

### Parallel Architectures

Because of its left-to-right nature, the *string chart* approach is most suitable for a pipeline architecture.

In [IPS91] we can find an implementation of CYK for an n-CUBE/7 with 64 processors using this approach. They make use of the possibility to simulate a one-way one-dimensional array of processors on a hypercube. The experimental results showed a maximum speedup of 33.1 using 64 processors. Nevertheless, there are several catches to these results. First, the above mentioned result was obtained with an input string of length 256. Speedup decreased considerably for smaller input strings (e.g. 10.6

$a_1$	0,1	0,2	0,3	0,4	0,5
	$a_2$	1,2	1,3	1,4	1,5
		$a_3$	2,3	2,4	2,5
			$a_4$	3,4	3,5
				$a_5$	4,5
					\$

Figure 4.3: Parse table layout.

for length 64). Second, the tests were conducted with a randomly generated grammar. Grammars for natural language usually have very specific properties; it is best to measure real-life performance by using a real-life grammar. Finally, the parser is not suitable for on-line parsing. Obviously, the parser has not been developed with natural language parsing in mind.

### 4.2.3 Tabular Chart Parsing

The tabular chart approach even further refines the data structure of the string chart parser. Tabular chart parsers arrange the item sets in a 2-dimensional fashion, as is shown in figure 4.3. Each table entry  $t_{i,j}$  can contain several items, each of which describes a possible parse for the part of the substring  $a_{i+1} \cdots a_j$ . First, the table entries  $t_{i,i+1}, i \geq 0$  are initialized. In subsequent steps, initialized table entries are used to compute other entries. This process continues until the entry  $t_{0,n}$  is filled, where  $n$  is the input string length. The big advantage of organizing the data this way is that the checking of items is reduced to a minimum. Moreover, the need of an internal agenda has vanished.

The exact dependencies of the  $t_{i,j}$  vary from method to method, but basically, a  $t_{i,j}$  is pair-wise dependent on  $t_{i,k}$  and  $t_{k,j}$  for  $i < k < j$ . This follows from the fact that consistent adjacent partial parses can be combined into one larger partial parse.

Just as with the string chart approach, we can assign a subset of the domain  $\mathcal{I}$  of some parsing method that specifies all possible primary items. The subset  $\mathcal{I}_{j,k} \subseteq \mathcal{I}$  of the items that can be deduced for the item set  $I_{j,k}$  of a table entry  $t_{j,k}$  is as follows:

$$\mathcal{I}_{j,k} = \{[\phi, j, k] \in \mathcal{I}\}, \text{ for each } k = 1, \dots, n, \quad (4.2)$$

where  $\phi$  can be anything, depending on the parsing method. As can be seen, each  $\mathcal{I}_{j,k}$



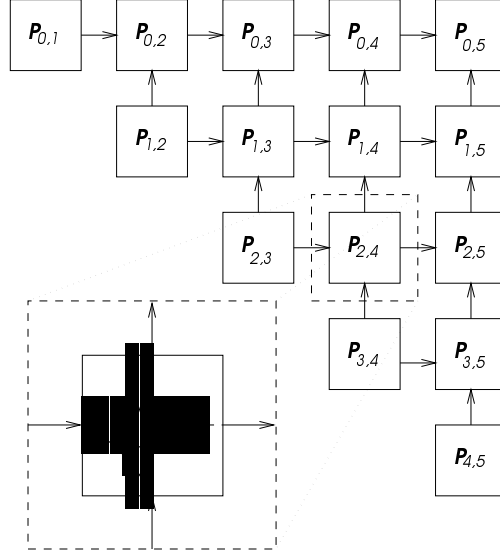


Figure 4.4: Processor assignment on a per entry basis. As is shown in the detailed depiction of processor 2,4, each processor forwards incoming messages from left to right, and from below upwards. New items that are derived within a cell, however, are sent both to the top and to the right.

is a subset of the  $\mathcal{I}_k$  that is used with the string chart approach. The parser accepts the input string if an item representing  $S \Rightarrow^* a_1 \dots a_n$  is contained in set  $I_{0,n}$ .

### Parallelization Techniques

The processor configuration that is typically used in these situations is displayed in figure 4.4. In this scenario, each process has full knowledge of the grammar.

Since the entries on one of the diagonals never depend on any of the other entries on the same diagonal, it is possible to compute the entries on each diagonal in parallel. [dV93a] describes several chart parsers that are parallelized on a ‘per-diagonal’ basis. In the general case, however, instead of waiting until an entire diagonal is completed before proceeding with the next, a more liberal filling order may be applied. The only restriction on the filling order is that the computation of all entries  $t_{k,l}$ , with  $l \leq j$  and  $k \geq i$  be completed before computing an entry  $t_{i,j}$ . This approach to parallelism normally results in parsers that parse in  $O(n)$  space per processor,  $O(n)$  time, and  $O(n^2)$  processors.

[Ni94] discusses several aspects of computation order in tabular chart parsers. It is noted that top-down filtering, as is implied, for example, by the deduction step  $D^{\text{Predict}}$  of the parsing method **E**, thwarts possible parallelism by opposing a left-to-right scheme. Intuitively, because of the predict deduction step, each new item that is deduced for, say, cell  $t_{i,j}$ , can cause the deduction of new items for cell  $t_{j,j}$ , which has a negative effect on the communication complexity. This, in turn, means that all cells in

one column are dependent on each other. The bottom-up parsing schemata **buE**, **CYK**, and **DD**, for example, do not have this characteristic. The init deduction step of these schemata can be completed before any other step. Cells only depend on cells in previous diagonals.

The computation order is also of influence on the space required per processor. At first, in absence of shared memory, the best approach seems to be letting each processor store each item it ever receives, so that the item will always be available immediately. At closer inspection, though, this scheme yields a space complexity of  $O(n^3)$  per processor. By choosing an appropriate order of computation, however, we can reduce this space complexity. The basic idea of the scheme described in [Nij94] is as follows. Each processor needs to compare items from one preceding cell in the same column to one preceding cell in the same row. Once all items of the two cells are compared, they can be discarded from the processor's memory. This kind of administration requires a highly synchronized cooperation amongst the processors; the order of computation is almost entirely predetermined. Since a full explanation of the computation order would be too extensive, we refer to [Nij94].

[HdV91] notes several facts that cause the implementation of parallel tabular parsers to be difficult:

- Item sets are small, which makes communication overhead dominate the computation time.
- The total amount of possible parallelism is not constant throughout the recognizing process. It ranges from  $O(n)$  at the start to  $O(n^2)$  halfway to  $O(n)$  at the end.
- There is a large variation in the required computation time per item set.

The last two difficulties seem to support the approach of dynamic load balancing.

**Example 4.1.** Algorithm 4.2.1 gives an example of a parallel tabular chart CYK parser. Each completed entry in the table contains a set of non-terminals such that  $A \in t_{i,j}$  if and only if  $A \Rightarrow^* a_{i+1} \cdots a_j$ . The input string  $x$  belongs to  $L(G)$  iff  $S \in t_{0,n}$ .

## Algorithm 4.2.1

(A Parallel CYK Recognizer)

**Input:** A context-free grammar  $G \in \mathcal{CNF}$  and a string  $x = a_1 \dots a_n$ .

**Output:** A parse table from which the parse tree can be extracted.

**begin**

1. **for all**  $t_{i,i+1}$ ,  $i = 0 \dots n - 1$  **pardo**
  - 1.1. Place each non-terminal  $A$  in  $t_{i,i+1}$ , for which there is a production  $A \rightarrow a_{i+1}$  in  $P$ .
2. **for all**  $t_{i,j}$ ,  $j - i > 1$  **pardo**
  - 2.1. **wait** until all entries  $t_{k,l}$ , with  $l \leq j$ ,  $k \geq i$  and  $(k, l) \neq (i, j)$  have been computed.
  - 2.2. Add any  $A \notin t_{i,j}$  to  $t_{i,j}$  if for any  $k$ ,  $i < k < j$ ,  $B \in t_{i,k}$ ,  $C \in t_{k,j}$  and  $A \rightarrow BC \in P$ .

end

A good overview of several sequential and parallel implementations of the CYK parser is given by [Nij90]. It includes several implementations of CYK-based chart parsers, both based on string and tabular versions.

### Parallel Architectures

A master slave with load balancing approach to tabular double dotted parsing (implementing **DD**) is described in [HdV91] and [OdV92]. Both describe an implementation for a Meiko with 16 INMOS T800 transputers. They conclude that the number of processors that can effectively be used is limited. ([HdV91] also describes a decentral approach for the Meiko which exhibited increasing speedup up till the maximum number of possible processors ( $n + 1$ ).) Both papers also conclude that tabular chart parsers do not seem to be appropriate for parallelization on the Meiko.

In [dV93b], we can find some additional measurements on parallel implementation of the CYK-, Earley, and Double Dotted, parsers. For all parsers, both traditional and context-sensitive variants were used. These experiments actually simulated parallelism, allowing an unlimited number of processors, and ignoring communication time. In spite of these favorable conditions, the best achievable speedup seemed to be of the order  $O(1)$ . In fact, the maximum overall speedup for *CYK* and *CYK*<sub>1,1</sub> was 4.3 resp. 4.6, for *E* and *E*<sub>1,1</sub> 3.3 resp. 3.1, and for *DD* and *DD*<sub>1,1</sub> resp. 4.2 resp. 3.5. On the other hand, the speedup that was gained with filtering bad items out seemed to improve speedup a great deal more.

In other words, from these experiments we might suspect that straightforward parallelization of chart parsers does not bring the solution to efficient context-free parsing. Amongst the reasons the author gives for this disappointing results is the fact that many processors stay idle, because usually only some cells on each diagonal need extensive computations. He suggests an island-driven approach might give better results.

Since the techniques described in 4.3 seem to achieve better results even though the underlying parsing schemes are essentially the same, it may be that efficiency can be achieved by considering subclasses of context-free languages.<sup>1</sup>

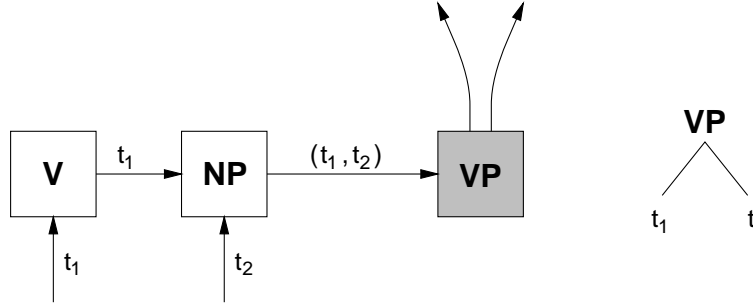
Finally, we would like to mention that the tabular chart approaches to parallelism seem to be appropriate for the design of VLSI implementations (see [CF82] for CYK, and [CF84], [Sij86], and [Tan83] for Earley).

## 4.3 Translating Grammar Rules into Process Configurations

It is also possible to take the individual grammar rules as units for parallelization. [YO94] describes a method in which each terminal and non-terminal of each production rule is represented by a separate process. Each process is referred to as an object, and can send and receive messages from other objects. Yonezawa and Ohsawa refer to it as an object-oriented parser.

---

<sup>1</sup>Hereby the author implicitly suggests that the better results of the techniques in 4.3 might partly be because of the limited grammar they can deal with (cycle- and  $\epsilon$ -free)

Figure 4.5: Objects representing the rule  $VP \rightarrow V NP$ 

The method is straightforward and can best be described by means of an example. Let us consider the following rule:

$$VP \rightarrow V NP \quad (4.3)$$

The interconnection of the objects associated with this rule is displayed in figure 4.5. Now, suppose that the object associated with  $V$  has received a partial parse tree  $t_1$  and that the object associated with the  $NP$  part of the rule has received the parse tree  $t_2$ . If the parts of the input string that are represented by  $t_1$  and  $t_2$  are adjacent to each other, the object labeled  $NP$  will forward the combined parse tree to the object labeled  $VP$ . This object, which represents the head of the given rule, will send the new parse tree to any occurrence of an object that represents a  $VP$  part on the right side of a production. This is also how  $NP$  and  $V$  originally received their messages. (Single input words (terminals) are also represented as parse trees; each terminal is also represented by an object.)

If (the object)  $NP$  finds that the parse trees  $t_1$  and  $t_2$  are not adjacent, it will simply store  $t_1$  in its local memory, for the case it will match with any arrival of a new  $t_2$ . If  $NP$  receives a  $t_2$  before  $t_1$ , it simply waits until a  $t_1$  is received.

Note that in case of ambiguity, the parser returns all possible parse trees. Yonezawa and Ohsawa derived a parallel time complexity of  $O(n \times h)$ , where  $h$  is the height of the parse tree, by means of a simulation. The reason for this is that there are example grammars that cause an explosion in the number of adjacency tests and the number of redundant subtrees that are generated, whereas grammars for natural languages usually do not cause these problems. [YO94] provides a detailed implementation of the parser (written in ABCL).

Unlike the chart approaches discussed in the previous section, Yonezawa and Ohsawa's parser implements a specific parsing method. Furthermore, as we did with the chart parsing approaches, we can define which processes can deduce which items. These are the topics of the following discussion.

### Relation to other Parsers

The parser is not able to handle the full range of context-free languages. The parser can only handle cycle- and  $\epsilon$ -free context-free languages. In the presence of cycles, the

parser might end up in an infinite loop. From the nature of bottom-up parsing follows that the method can only handle  $\epsilon$ -free languages. The latter, however, is not a real problem, because all grammars containing  $\epsilon$ -rules can be transformed in an equivalent  $\epsilon$ -free grammar.

It is interesting to compare this approach with the chart parsing approaches, and to relate it to the theory discussed in chapter 3. In [YO94], it is not mentioned which particular parsing method underlies this parser. It is straightforward, however, to express the method in terms of items, because the information that is sent from object to object is closely related to the items used in parsing schema.

Let us first consider the hypotheses. The hypotheses are the same as with most other parsing schemata, viz. each input symbol is represented with an item of the form  $[a_i, i - 1, i]$ . Initially, these hypotheses are sent to objects representing the respective symbol (all possible symbols are represented as objects in the network). As explained before, the left most object of the right hand (type-2) of a production simply passes the item to the next adjacent object. The objects representing the left-hand of a production (type-1) have the function of broadcasting matches to the appropriate objects. So to investigate what production rules underlie the YO (for Yonezawa and Ohsawa) parser, we need to investigate the functioning of the objects representing all but the first right-hand production symbols (type-3).

Consider, for example, a type-3 object representing  $NP$  in

$$VP \rightarrow VNP. \quad (4.4)$$

This object will only produce a new item if it receives an item representing a partial parse tree for  $NP$  which is adjacent to some item received from the left-hand object representing  $V$ . This behavior can be represented by the following deduction rule, similar to Earley parsing:

$$[A \rightarrow \alpha \bullet X \beta], [X \rightarrow \gamma \bullet] \vdash [A \rightarrow \alpha X \bullet \beta]$$

Note that recognition only occurs from left to right. A type-3 object will only receive (and produce) items representing a partial recognition starting at the complete left of the production. This is essentially bottom-up Earley (**buE**), The rule encompasses both  $D^{\text{scan}}$  and  $D^{\text{complete}}$ ; the parser does not distinguish between terminals and non-terminals. The parser does not have to initialize by introducing items of the form  $[A \rightarrow \bullet \gamma, i, i]$ , because these are actually represented by the topology of the network of objects itself. So, regarding the domain of items that are passed, this method closely resembles **buLC** (which is, in fact, almost identical to **buE**).

The fact that there needs to be an object for each terminal, as if it were a non-terminal, suggests that the grammar for this method is in fact  $\mathcal{CNF}$ , but allowing more than two non-terminals in a row.

The essential difference with chart parsers is the way in which the work is divided over processes and the items are spread over the network. As a result, items of the form  $[A \rightarrow \alpha \bullet \beta]$ ,  $\alpha, \beta \neq \epsilon$ , are only managed by the processes representing the respective production. Only items of the form  $[A \rightarrow \alpha \bullet]$  need to be broadcasted to any type-2 or type-3 object representing  $A$ .

To define the set of items that can be deduced by an object, we first assume, for notational reasons, that each object that represents a right-hand symbol in a production

is assigned an unique id. For example, if we write  $A \rightarrow \alpha B_i \beta \in P$  we indicate the production in  $P$  of which object  $i$  is part of. We will call the set of all ids  $\mathcal{N}$ . Now, the subset  $\mathcal{I}_k \subseteq \mathcal{I}$  of the items that can be deduced by an object with id  $k$  is as follows:

$$\mathcal{I}_k = \{[A \rightarrow \alpha B_k \bullet \beta, i, j] \in \mathcal{I}_{YO}\}, \text{ for each } k \in \mathcal{N} \quad (4.5)$$

In contrast, a tabular chart parser, for example, distinguishes processes by for which range of the input string they can recognize partial parses. Note, however, that the items that are used in [YO94] are different from Earley-like items.

### Characteristics

According to Yonezawa and Oshawa, their parser has several advantages over tabular parsers. To start with, the parser is well suited for on-line parsing. The words need not be provided in any particular order, and above all, any utterance can (partially) be retracted at any time during the parse. This is implemented by means of “anti-messages”, which have a similar high-priority status as exceptions.

Furthermore, the parser is capable of parsing more than one sentence at a time. This is useful if one considers a scenario where each object is running on a separate processor. By parsing more than one sentence at a time, more efficient use of the processors is made.

Finally, it is possible to integrate semantic checking into the parser without much effort. [YO94] describes a way of incorporating semantic checking into the parser in a pipelined fashion. According to the authors this is a strong plus, comparing the parsing scheme to competing schemes as CYK and Earley.

[YO94] notes that the advantages of the object-oriented approach could perhaps also be obtained by the CYK scheme, and similar schemes, if the individual components are implemented as concurrent components capable of transmitting messages.

### Parallel Architecture

Considering the fact that grammars may contain many rules, this method is suitable for massively parallel machines. Until now, however, massively parallel execution has only been researched by means of simulation.

## 4.4 Connectionist Parsing Algorithms

There have been several proposals for Connectionist approaches to the parsing problem. They all have in common that there is some network of simple elements, or nodes, which function without any central control. The nodes are connected by weighted links. Each node computes a function that is dependent on the weighted inputs and the current activation level of the node. [Ni91] provides an overview of several of those proposals.

Usually, each node represents a syntactical category (‘localist view’). One of the common problems that occurs with connectionist approaches is that a fixed size of the network usually implies a maximum length of the input string. Therefore, some approaches allow the network to be built at the time a string is parsed.

According to [Nij91], none of the connectionist context-free parsers provides a natural solution to the parsing problem. Therefore, we will only touch upon the general idea of context-free connectionist parsing.

There is no need to explicitly mention the parallelization techniques and parallel architectures, since these are implied by the connectionist approach. In other words, all of the methods described in this section follow the spreading activation approach, as mentioned in section 2.4.

### 4.4.1 Connectionist CYK

[Fan94] describes a simple connectionist network which is based on **CYK**. It is sound and complete (except for  $\epsilon$ -productions). It can easily be expanded to  $\epsilon$ -free context-free grammars, but [Fan94] warns for an increase in space complexity if this is done. It is closely related to the CYK parser, which makes it interesting to compare it with the chart parsing techniques described in section 4.2.

The nodes of the network are arranged in a way that resembles the upper triangular parse matrix of the tabular CYK parser. Usually, a non-terminal  $A$  is added to some cell  $t_{i,j}$  of the matrix, if there is some production  $A \rightarrow BC \in P$  and  $B \in t_{i,k}$  and  $C \in t_{k,j}$ . In Fanty's network, cell  $t_{i,j}$  already contains a node resembling  $A$  which becomes active when it receives an activation input from both a node representing  $B$  in  $t_{i,k}$  and a node representing  $C$  in  $t_{k,j}$ . Their are inputs for each possible pair of cells that yield a correct (partial) parse for  $A$ , by any production rule. If both inputs of any pair receive a firing signal, the node will fire itself.

A similar scheme applies to the recognition of terminals. Terminals are recognized at the diagonal for which  $j - i = 1$ ; non-terminals are recognized at the diagonals for which  $j - i \geq 2$ .

The network so far yields a (bottom-up) recognizer. [Fan94] also describes a technique for finding the actual parse trees. To accommodate this feature, each node (each match, non-terminal, and terminal node) is augmented with a top-down counterpart. As soon as the bottom-up non-terminal node for  $S$  in the upper-right corner is recognized, it will activate its top-down counterpart, which in turn will start propagating activation signals top-down. Ultimately, activated top-down nodes will represent parse trees.

A top-down node is activated if it receives an activation from its bottom-up counterpart (as soon as the bottom-up counterpart is activated) and any other source, indicating it has been used higher in the hierarchy to form a complete parse. Since bottom-up match units will only get activated when all inputs have received an activation signal, a top-down node can never be activated if it does not represent a correct (partial) parse. Notice how ambiguity is represented by multiple active top-down non-terminal nodes within one cell.

Constructing the network is straightforward. Each cell  $t_{i,i}$  contains all terminals. The cells  $t_{i,j}$ , for which  $j - i = 1$ , contain nodes (match and terminal) to represent the productions in  $P$  of the form  $A \rightarrow a$ . On each further diagonal, the cells  $t_{i,j}$  contain nodes to represent all non-terminals, provided that a non-terminal can produce a string of length  $j - i$ . Each cell on a certain diagonal will contain the same set of nodes. [Nij91] refers to the constructing of the network as meta-CYK-parsing.

The resulting network uses  $O(n^{m+1})$  space, where  $n$  is the length of the network,

and  $m$  the number of non-terminals on the right-hand side of the productions. It computes a parse in  $O(n)$  parallel time (serial time  $O(n^3)$ ), where  $n$  is the length of the input string.

### Parallel Architectures

The large number of nodes that are required for the network indicate that this method is best suited for massively parallel machines.

### Characteristics

This connectionist approach allows easy extension to deal with ambiguity. Amongst other methods, [Fan94] describes a solution using inhibitory links, and suggest such a network could be trained to let it prefer more common interpretations.

The network also allows easy extension to handle near-miss input, provided that the incorrect part is of the same length as some correct part. For other cases, according to [Fan94], the network is too inflexible.

Finally, [Fan94] describes some extensions to facilitate the network with learning capabilities, including local learning, global learning, and deferred learning.

A major disadvantage of the network is that it needs to be rebuild for any new input length, which makes it highly unsuitable for on-line parsing.

#### 4.4.2 Other Connectionist Approaches

There are many more connectionist methods. However, as we mentioned before, they do not seem to give satisfying solutions to the context-free parsing problems. Most importantly, it seems that the context-free connectionist approach is problematic if it comes to on-line parsing.

We will only mention some of the other connectionist approaches to context-free parsing. First of all, [Sel94] describes a parallel connectionist parsing based on the Boltzmann machine, i.e., parallel stochastic relaxation using simulated annealing. [How88] also describes a relaxation algorithm, which uses decay over time together with a competition for available activation. Finally, [NM88] describes a connectionist parallel left-corner parser.

For completeness, we mention that there are some complete different approaches to connectionism for natural language processing that are not based on syntactic analysis, but rather use techniques like strongly interactive distributed processing of word senses, case roles, and semantic markers (from [odAANL89], see e.g. [CS84], [WP85], and [MK86]). Obviously, these approaches are outside the scope of this report.

## 4.5 Reducing the Parsing Problem to Other Problems

### 4.5.1 Parsing as Matrix-Multiplication

[Tho94] rewords the fact that a basic step in a tabular chart parser is functionally equivalent to boolean matrix multiplication. Parallel matrix multiplication is a much



$$\left( B : \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \times C : \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \right) \cup A : \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \Rightarrow A' : \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Figure 4.6: Matrix multiplication to recognize  $A \rightarrow BC$ , where the 1 in  $A'$  resembles  $A \Rightarrow^* a_1 \dots a_2$

studied issue, so it can be fruitful to investigate this relation in more depth.

We consider a parser for a grammar  $\mathcal{CNF}^+$ , which is  $\mathcal{CNF}$  augmented with productions of the form  $A \rightarrow B, B \in N$ . Recognition of this grammar can be translated into matrix multiplication as follows. Each non-terminal is associated an  $(n+1) \times (n+1)$  matrix (the indices do not indicate symbol positions, but inter-symbol positions, starting left from the left most symbol (0) to right from the right most symbol ( $n$ )), where  $n$  is the length of the input string. An entry  $c_{i,j}$  in a matrix for non-terminal  $A$  is **true** if and only if  $A \Rightarrow^* a_{i+1} \dots a_j$ .

Finding adjacent constituents for a production of the form  $A \rightarrow BC$ , reduces to multiplying the matrices representing the right-hand of the production,  $B \times C$ , and assigning the element-wise disjunction of the result and the matrix for  $A$  to  $A$ . This process is illustrated in figure 4.6. A complete recognizer is given by algorithm 4.5.1. [Tho94] also describes a modification to make it useful as a parser. The basic idea is to record which adjacent parts, and which production, caused an entry to be set to true. This can be achieved by recording the production number and the position at which the two parts adjoined, in each cell that is set to true. (the latter only in the case of a production of the form  $A \rightarrow BC$ ).

### Algorithm 4.5.1

(Boolean Matrix Multiplication Recognizing)

**Input:** A grammar  $G \in \mathcal{CNF}^+$ , and an input string  $x = a_1 \dots a_n$ .

**Output:** Returns **true** if the string is recognized, else **false**.

**begin**

1. Create a  $(n+1) \times (n+1)$  matrix  $M^A$  for all  $A \in N$ , and set all entries  $M_{i,j}^A$  to **false**.
2. For all  $A$  and  $a_i$ , for which  $A \rightarrow a \in P \wedge a_i = a$ , set  $M_{i-1,i}^A = \mathbf{true}$ .
3. For each  $A \rightarrow \epsilon \in P$ , set  $M_{i,i}^A = \mathbf{true}$ .
4. **while** iterations yield changes **do**
  - 4.1. For each  $A \rightarrow B \in P$  set  $M_{i,j}^A = M_{i,j}^A \vee M_{i,j}^B$ .
  - 4.2. For each  $A \rightarrow BC \in P$  set  $M_{i,j}^A = M_{i,j}^A \vee (M_{i,j}^B \wedge M_{i,j}^C)$ .
5. **return**  $M_{0,n}^S$ .

**end**

### Parallel Architecture

[Tho94] gives experimental results of an implementation for the Connection Machine. The Connection Machine is a large scale SIMD processor. On a systolic array, matrix multiplication can be carried out in  $O(n)$  time. It is, however, straightforward to simulate a systolic array on the connection machine. So the results of the experiments are to some extent representative for systolic arrays as well.

Let us now take a closer look at algorithm. Each time  $O(m)$  multiplications are needed, where  $m$  is the number of productions in  $P$ . Depending on the depth of the parse tree, the number of loops varies from  $\log_2 n$  to  $n$ , with a worst case of  $O(n)$ . This amounts to a total time complexity of  $O(n^2m)$ . The matrices for each non-terminal, however, can be executed in parallel as well, reducing the factor  $m$  to  $m'$ , the maximum number of production rules per non-terminal. [Tho94] also applies a bottom-up filter technique to reduce the number of multiplications.

#### 4.5.2 Context-Free Parsing as Parallel Logic Programming

The *definite clause grammar* formalism can be thought of, roughly, as a context-free grammar formalism, augmented with the possibilities of specifying constraints in Prolog and allowing attributed non-terminals. [Mat86] describes a parser for Prolog, based on the DCG formalism. Instead of using the usual recursive descent top-down parser, which is found in most Prolog implementations, it is a bottom-up parser which is based on the left-corner method. [Mat86] argues that the main advantage of implementing a context-free parser as a DCG parser is the ease with which extensions and changes can be applied.

In Matsumoto's approach, the parser keeps track of the sequence of rules, and especially the positions within these rules, reached so far, by making use of identifiers. Before a context-free grammar is converted to Prolog, each position between two symbols on the right-hand side is marked with a unique identifier. How this is used in the actual algorithm can best be described by means of an example (a slightly simplified (and incomplete) version of [Mat86]). Suppose we have the following context-free grammar.

$$\begin{aligned} S &\rightarrow NP_1 VP \\ NP &\rightarrow *det_2 *noun \\ NP &\rightarrow *det_3 *noun_4 *relc \\ VP &\rightarrow *verb \\ VP &\rightarrow *verb_5 NP \end{aligned}$$

We already assigned unique identifiers to all positions between two subsequent symbols in the the right-hand sides of the productions. Skipping the straightforward conversion to a DCG grammar, this grammar can be translated into Prolog as follows

```
np(X, id1(X)).
vp(id1(X), Y) :- s(X, Y).
```

```

det(X,id2(X)).
noun(id2(X),Y) :- np(X,Y).

det(X,id3(X)).
noun(id3(X),id4(X)).
relc(id4(X),Y) :- np(X,Y).

verb(X,Y) :- vp(X,Y).

verb(X,id5(X)).
np(id5(X),Y) :- vp(X,Y).

the(X,Y) :- det(X,Y).
man(X,Y) :- noun(X,Y).
walks(X,Y) :- verb(X,Y).

```

The heads of clauses representing a left corner of a production (type 1) have a variable as its first argument. For all other clauses (type 2), the first argument of the head requires the functor of the structure being passed to be the identifier that precedes the symbol in the production represented by the respective clause.

If we augment the Prolog program with the clause

```
s(begin,end).
```

we can see whether parsing the sentence ‘the man walks’ will be recognized by this grammar (it will) by calling

```
the(begin,X), man(X,Y), walks(Y,end).
```

The matching of the first word will result in a value `id2(begin)` for `X`. This indicates that the parser started to attempt to complete a noun phrase (`np`). This noun phrase can be completed with the second word. On completion of a production, the identifier that was originally attached to the structure may be removed. A new identifier is introduced (`Y=id1(begin)`) indicating that an attempt is made to parse `s`. The third word completes the parse.

### Parallelization Technique

For the parallel implementation a slightly different approach is taken, both to avoid superfluous computation and to enable an efficient parallel computation scheme. The idea is to pass, instead of one followed path, a set of all paths that have been followed so far. Having a set of followed paths, instead of a list, prevents that some paths are followed multiple times if some paths happen to coincide.

We will give a conversion of several parts of the original Prolog program into Parlog clauses to clarify this scheme. The two clauses with head `det` are translated as follows.

```

mode det1(?,^).
det1(X,[id2(X),id3(X)]).

```

The list in the second argument indicates the possibilities for further parsing upon parsing a `det`. The clauses for `verb` can be translated similarly.

```

mode verb1(?,^)
verb1(X,[id6(X)|Y]) :- |
    vp(X,Y).

```

Type 2 clauses are a little bit more complicated to translate. We need to separate each case. For `noun` we proceed as follows.

```

mode noun2(?,^)
noun2([],[]).
noun2([id2(X)|T],Y) :- |
    np(X,Y2),
    noun2(T,Y1),
    merge(Y1,Y2,Y).
noun2([id3(X)|T,[id4(X)|Y]) :- |
    noun2(T,Y);
noun2([_|X],Y) :- |
    noun2(X,Y).

```

The recursive call to `noun2` ensures that all paths in the list are processed. The call to `merge` merges the paths returned by this call and the path associated with the specific clause.

Type 1 and type 2 clauses for a specific non-terminal still need to be combined into a single operation. This is straightforward and is accomplished by merging the type 1 and type 2 results.

Unfortunately, merging is a rather inefficient operation. In Prolog, difference lists can be used to obtain a more efficient program. Difference lists can also be used with Parlog, but will restrict the order of computation.

We will not go into any details about the possible strategies of parallel prolog implementations (see instead section 2.4), because these are more appropriately discussed in a subsequent report on parallel attribute matching.

### Relation to Other Parsers

The parser works basically the same as a basic chart parser. The big difference is that the chart parser needs to perform adjacency tests to find matching items, whereas Matsumoto's parser finds matching items by means of shared variables.

The parallel time complexity of the parser is proportional to the depth of the parse tree, so  $O(n)$ . Matsumoto concludes that the space complexity is not larger than that of a regular (basic) chart parser, because the parser cannot create more items.

### Parallel Architecture

The architecture for which this approach is suitable really depends on for which architecture some parallel implementation of Prolog is suitable.

## Chapter 5

# Suggestions for Future Research

A useful parser for practical natural language programming should be able to deal with both the syntactic and semantic aspects of language. In this report, we have given a survey of parallel parsing techniques for context-free grammars, which can serve as the backbone for a natural language parser. We will need to investigate extensions to context-free parsers that are both powerful enough to deal with all the requirements and efficient enough for practical use.

The ultimate goal, however, is to obtain a practical and efficient dialog system. Considering the vast amount of different approaches and the lack of theory to determine which of these approaches is best suited for any specific situation, we need to look closely at existing systems that are close to our own requirements. Therefore, before we will give the final suggestions for further research, we will first discuss some related research.

First, we will discuss some entirely different approaches to parsing than the one we follow. After that, to give an indication of the wide variety of methods that are used in practice, we continue with some brief examples of actual NLP systems, and mention some of the essential techniques that were adopted for these systems.

We conclude this chapter with some notes of importance concerning context-free parsing in relation to our project (most notably regarding the n-CUBE architecture), and the suggestions for further research itself.

### 5.1 Other Approaches to Parsing

As mentioned before, we have deliberately omitted the topic of parsing attribute grammars, because we wanted to focus on context-free grammars. Both classes of grammars are phrase structure grammars. However, by focusing solely on context-free grammars, we also omitted the coverage of other parsing methods that do not fall into the class of phrase structure grammars. Without the intention of being complete, we will now describe several other classes that are also used for natural language parsing. This taxonomy has been taken from [Nag96].

**PSG** Once more, for completeness, we will give a short description of *phrase structure grammars*, which include context-free grammars, but also the whole range of attribute grammars, HPSG, ATN, Tree Adjoining Grammars, etc. Phrase structure grammars, in fact, define a language by specifying how each sentence of that language may be generated.

The disadvantage of phrase structure grammars is that it is virtually impossible to create a grammar that completely covers a certain natural language, especially if this is to be done by hand. The advantage, however, is that a parse tree, which generally is obtained as a result, contains a lot of useful information for analysis.

**DG** Instead of defining which sentences can be generated, a *dependency grammar* defines a way to interpret a given sentence by finding modifier-modifiee relations. No assumptions are made about a particular structure of the sentence; it just determines which word modifies which other word. So whereas PSGs basically define a language, DGs merely analyze a sentence.

DGs have the disadvantage that they are weaker than PSG. The advantage, however, is that they always return a full analysis, and hence are very robust.

**CG** *Case grammars* interpret a sentence by filling in slots of a case frame. A case frame can be seen as a meaning representation of a sentence, related to some word. The frames are filled in by considering the meaning and function of the words in the sentence.

The disadvantage is, for example, that a grammar requires many semantic markers to be specified for each possible usage of a verb.

Despite of the robustness problems, PSGs still seem to be most appropriate for Germanic languages, because phrase structure is an essential element of these languages; by omitting an analysis of the phrase structure, useful information is lost.

At this point, we should also mention additional strategies for (parallel) context-free parsing, being probabilistic parsing and ordering grammars. Especially probabilistic parsing has gained interest in recent research.

## 5.2 Existing Natural Language Processing Systems

In this section, we will briefly discuss a preliminary selection of existing natural language systems (a more comprehensive and complete overview of existing systems remains a topic of further research). With the current state of the art of computational linguistics it is impossible to determine the best approach for the design of a NLP system from theoretical knowledge only. In other words, the development of practical natural language processing systems usually requires an experimental and data-driven approach ([Jen91]). It will, therefore, undoubtedly be fruitful to consider the approaches of other projects, because this may show the practical applicability or uselessness of certain techniques.

Besides systems for dialog systems, it is also useful to consider systems for machine translation and knowledge acquisition, because they often use parsing methods fitting our own requirements. Furthermore, it is also useful to consider systems not specifically designed for parallel architectures, because it may prove which techniques can be

successfully embedded in a practical natural language system. Moreover, many of the modern systems have an inherent parallelism incorporated in their design because of psycho-linguistic reasons. It may be fruitful to investigate to what extent this inherent parallelism can be explored on an actual parallel computer.

**PROTEUS** Proteus is an ongoing research project concerned with a wide range of NLP applications, including information retrieval, information extraction, machine translation, and language modeling for speech recognition.

[GC88] discusses the parser intended for the PROTEUS system. PROTEUS uses a context-free grammar, augmented with the capability of handling procedural restrictions for semantic and syntactic constraints. It is essentially a basic chart approach implementing a Earley(**E**)-like method. The parser is intended for a shared memory system with a limited number of processors. (As discussed before, the basic chart approach is limited to such systems.)

**Whiteboard Architecture** [BS94] describes a distributed architecture where separate components notify their (incremental) results to a separate *coordinator* which keeps track of the data and coordinates all computations. This system tries to overcome the problems of blackboard architectures (where components write their intermediate results in a shared data structure), solving problems as control of concurrent access and extensive communication loads.

The rudimentary prototype uses an island-driven syntactic chart-parser.

**ATLAST** A typical example of an architecture with inherent concurrency based on psycho-linguistic arguments is ATLAST (see [ERHG94]). The system is made up of three independent components, which operate concurrently to find a single representation of the input string: a syntactic analyzer, a semantic analyzer, and an evaluation mechanism that constructs the final representation. The most interesting property of this system is that, contrary to the common syntax first approach, the semantic analyzer does not depend on output of the syntactic analyzer, and vice versa: the system can determine semantics for ungrammatical sentences and can recognize syntactic validity independent of whether a sentence is semantically sane. The two components do, however, limit each other's search space.

The syntactic analyzer is based on a simple augmented transition network (ATN) parser. Furthermore, all of the components take a spreading of activation approach, hence parallelization is straightforward.

**PEP** The *Parallel Expert Parser* discussed in [DA94] defines a framework in which different linguistic components can be combined. It is a descendent of WEP, the Word Expert Parser, with the aim of making this system suitable for parallel systems. The PEP approach solves the problem of the problematic communication bottleneck that was inherent to the WEP approach. Each implementation of a component, or expert, need be compiled into Flat Concurrent Prolog, the adopted language of PEP. [DA94] describes several implementation aspects for different parallel Prolog systems. These approaches are appropriate for a course-grained parallelism, although [DA94] notes the possibility of making it more fine grained by integrating the system with fine-grained methods.

The parser is not specifically syntactic or semantic based, but rather is claimed to be an *integrated parser*. It defines a rigid communication interface for the several components, which can contain any kind of expert knowledge. However, from the definition of the system can be concluded that the system is more meaning than syntax oriented.

Although we did not go into any details, it may have become clear that we can find a wide range of different systems suitable for different architectures, and using complete different techniques. We see, for example, that parallelization is pursued by parallelizing conventional parsing methods, by using inherent parallelism in specific architectures, or by connectionist approaches. Also, many different grammar formalisms are used.

### 5.3 Context-Free Parsing and the IMPACT Project

In the introduction we mentioned that we would investigate whether the n-CUBE can effectively be used for the implementation of the dialog system. From [Tho94] it appears that the n-CUBE is not very suitable for context-free parsing using the basic chart approach.

Another approach of context-free parsing on the hypercube, given in [IPS91], seems to give better results. Nevertheless, the experimental results of the latter approach were not based on grammars for natural language. It remains to be seen whether the positive results will hold if grammars for natural languages are used.

From several experiments conducted within different researches (most notably [HdV91] and [IPS91]), we can conclude that load balancing is the better approach to chart parsing.

### 5.4 Directions for Further Research

As was mentioned several times before in this paper, we will shift our attention to parallel attribute evaluation in the near future. Nevertheless, keeping in mind our original goal—an efficient and practical dialog system—and considering what we have discussed in this report, we suggest that the following points should also receive attention in the near future.

In deciding which techniques to adopt for the final dialog system, it will probably be profitable to look at the experimental results of other, related projects. In other words, investigating approaches used in other systems may teach us which specific architectures and techniques are most suitable for our own project. In addition, it will probably be difficult to design an approach that will be optimal for both small- and large-scale parallelism. It is therefore important to choose for which architectures one intends to develop the system. On the other hand, it is still desirable that the design is as generally applicable as possible. Therefore, we will first focus our research on architecture independent techniques. In particular, we will, for the time being, focus on the following topics:

- measuring the communication complexity of different parallelized versions of the available sequential system, based on existing grammars;



- a more in-dept investigation of existing natural language processing systems;
- parallelizing parsers for attribute-value grammars.

Finally, as a guideline for continuing the development of the parallel parser, we would like to mention again the findings discussed in [dV93b]. In this paper it was shown that introducing enhancements like look-ahead can often give more speedup than straightforward parallelization of the algorithm. Whatever the case may be, we need to closely investigate where the best possibilities for parallelization lie, regarding all components of the system.

[RS97, BT96]



# References

- [AH94] Geert Adriaens and Udo Hahn, editors. *Parallel Natural Language Processing*. Ablex Publishing Corporation, Norwood, New Jersey, 1994. 5, 7, 7, 7, 7, 25, 49, 50, 50, 51, 52, 52, 53
- [BS94] Christian Boitet and Mark Seligman. The “whiteboard” architecture: A way to integrate heterogeneous components of nlp systems. In *Fifteenth International Conference on Computational Linguistics (COLING’94)*, Kyoto, Japan, August 1994. 45
- [BT96] Harry Bunt and Masaru Tomita, editors. *Recent Advances in Parsing Technology*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996. 47, 51, 52
- [CF82] K. H. Chu and K. S. Fu. VLSI architectures for high-speed recognition of context-free languages and finite-state languages. In *Proceedings of the Ninth Annual Symposium on Computer Architectures*, 1982. Also in SIGARCH Newsletter, 10, 3, 43–49. 33
- [CF84] Y. T. Chiang and K. S. Fu. Parsing algorithms and VLSI implementations for syntactic pattern recognition. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 3 of PAMI-6, pages 302–314, 1984. 33
- [CS84] G. W. Cottrell and S. L. Small. Viewing parsing as word sense discrimination: a connectionist approach. In *Computational Models of Natural Language Processing*, pages 91–119. North-Holland, Amsterdam, 1984. 38
- [DA94] Mark Devos and Geert Adriaens. The parallel expert parser. In Adriaens and Hahn [AH94]. 45, 45, 45
- [DKM84] Cynthia Dwork, Paris Kanellakis, and John Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984. 7
- [dV93a] J.P.M. de Vreught. *Parallel Parsing*. PhD thesis, Delft University of Technology, Delft, 1993. 4, 31

- [dV93b] J.P.M. de Vreught. A practical comparison between parallel tabular recognizers. In *Parsing natural language. Proceedings of the sixth Twente Workshop on Language Technology (TWLT6)*, pages 63–70, 1993. 33, 47
- [dVH89] J.P.M. de Vreught and H.J. Honig. A tabular bottom-up recognizer. Technical Report 90-31, Delft University of Technology, dept. Computer Science, Delft, Netherlands, 1989. 18, 19
- [dVH91] J.P.M. de Vreught and H.J. Honig. Slow and fast parallel recognition. In *2nd International Workshop on Parsing Technologies (TWLT)*, Cancun, Mexico, 1991. 18
- [ERHG94] Kurt P. Eiselt and Jr. Richard H. Granger. Process independence and concurrency in a model of sentence understanding. In Adriaens and Hahn [AH94]. 45
- [Fan94] Mark Fanty. Context-free parsing in connectionist networks. In Adriaens and Hahn [AH94]. 37, 37, 37, 38, 38, 38
- [Fis75] C.N. Fischer. *Parsing context-free languages in parallel environments*. (tech. rep. no. 75-237). unpublished doctoral dissertation, Cornell University, Dept. of Computer Science, Ithaca, NY, 1975. 26
- [GC88] R. Grisham and M. Chitrao. Evaluation of a parallel chart parser. In *Second Conf. on Applied Natural Language Processing*, pages 71–76. Association for Computational Linguistics, 9–12 February 1988. 28, 45
- [GR88] A. Gibbons and W. Rytter. Parallel recognition and parsing of context-free languages. In *Efficient parallel algorithms*. Cambridge University Press, 1988. 25
- [HdV91] J. Hoogerbrugge and J.P.M. de Vreught. Parallel recognizing in practice. Technical Report 91-26, Delft University of Technology, dept. Computer Science, Delft, Netherlands, 1991. 32, 33, 33, 46
- [How88] T. Howells. Vital: A connectionist parser. In *Proceedings of the Tenth Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum, 1988. 38
- [IPS91] Ibarra, Pong, and Sohn. Parallel recognition and parsing on the hypercube. *IEEE Transactions on Computers*, 40(6):764–770, 1991. 29, 46, 46
- [Jen91] Karen Jensen. A broad-coverage natural language analysis system. In Tomita [Tom91], chapter 17. 44
- [Kac90] Péter Kacsuk. *Execution Models of Prolog for Parallel Computers*. Research monographs in parallel and distributed computing. MIT Press, Cambridge, Ma., 1990. 7

- [Mat86] Yuji Matsumoto. A parallel parsing system for natural language analysis. In *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science 225, pages 396–409, Berlin, 1986. Springer-Verlag. 40, 40, 40
- [MK86] James L. McClelland and A. H. Kawamoto. Mechanisms of sentence processing: assigning roles to constituents of sentences. In David E. Rumelhart James L. McClelland and the PDP Research Group, editors, *Parallel Distributed Processing*, volume 2, chapter 19, pages 272–325. MIT Press, Cambridge, Ma., 1986. 38
- [MTH<sup>+</sup>83] Y. Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. Bup: A bottom-up parser embedded in prolog. In *New Generation Computing*, volume 1, pages 145–158, 1983. 20
- [Nag96] Makoto Nagao. Varieties of heuristics in sentence parsing. In Bunt and Tomita [BT96]. 43
- [Ned93] Mark-Jan Nederhof. Generalized left-corner parsing. In *6th Meeting of the European Association of Computational Linguistics (ACL)*, pages 305–314, Utrecht, Netherlands, 1993. 20
- [NH96] Peter Neuhaus and Udo Hahn. [Trading off completeness for efficiency—the parsetalk performance grammar approach to real-world text parsing.](#) In *Proceedings of FLAIRS’96*, Key West, 1996. 1
- [Nij90] Anton Nijholt. The cyk-approach to serial and parallel parsing. Memoranda Informatica 90-13, University of Twente, dept. of Computer Science, March 1990. 33
- [Nij91] A. Nijholt. Overview of parallel parsing strategies. In Tomita [Tom91], chapter 14. 8, 25, 26, 26, 36, 37, 37
- [Nij94] Anton Nijholt. Parallel approaches to context-free language parsing. In Adriaens and Hahn [AH94]. 26, 27, 27, 28, 31, 32, 32
- [NM88] H. Nakagawa and T. Mori. A parser based on a connectionist model. In *Proceedings of the Twelfth International Conference on Computational Linguistics*, COLING 88, pages 454–458, Budapest, 1988. 38
- [odAANL89] R. op den Akker, H. Alblas, A. Nijholt, and P.H.W.M. Oude Luttighuis. An annotated bibliography on parallel parsing. Technical Report INF89-67, Faculteit der Informatica, Universiteit Twente, December 1989. 38
- [OdV92] J.G.E. Olk and J.P.M. de Vreught. Evaluating master-slave implementation of double-dot parsing for massive parallel mind systems. Technical Report 92-78, Delft University of Technology, dept. Computer Science, Delft, Netherlands, 1992. 33
- [RL70] D.J. Rosenkrantz and P.M. Lewis. Deterministic left corner parsing. In *11th Annual Symposium on Switching and Automata Theory*, pages 139–152, 1970. 20

- [RS97] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages (Vol 2) Linear Modeling: Background and Application*. Springer Verlag, Berlin, 1997. 47, 52
- [Sel94] Bart Selman. Parsing as an energy minimization problem. In Adriaens and Hahn [AH94]. 38
- [Sij86] F. W. Sijsterman. Parallel parsing of context-free languages. Esprit Project 415 Doc. No. 202, Philips Research Laboratories, Eindhoven, 1986. Subproject A: Object-oriented language approach. 33
- [Sik93a] K. Sikkel. On-line parsing in constant time per word. *Theoretical Computer Science*, 120:303–310, 1993. 15, 15
- [Sik93b] Klaas Sikkel. *Parsing Schemata*. PhD thesis, Dept. of Computer Science, University of Twente, Enschede, The Netherlands, 1993. 9, 27
- [SN97] Klaas Sikkel and Anton Nijholt. Parsing of context-free languages. In Rozenberg and Salomaa [RS97]. 9, 10, 11, 12, 18, 19
- [SodA96] Klaas Sikkel and Rieks op den Akker. Predictive head-corner chart parsing. In Bunt and Tomita [BT96]. 22
- [Tan83] H. D. A. Tan. VLSI-algoritmen voor herkenning van context-vrije talen in lineaire tijd. Technical Report IN 24/83, Stichting Mathematisch Centrum, Amsterdam, 1983. 33
- [Tho91] Henry S. Thompson. Chart parsing for loosely coupled parallel systems. In Tomita [Tom91], chapter 15. 28, 28, 28
- [Tho94] Henry S. Thompson. Parallel parsers for context-free grammars—two actual implementations compared. In Adriaens and Hahn [AH94]. 28, 38, 39, 40, 40, 46
- [Tom91] M. Tomita, editor. *Current Issues in Parsing Technology*. Kluwer Academic Publishers, Norwell, MA, 1991. 26, 50, 51, 52
- [vdBAK<sup>+</sup>96] S.P. van de Burgt, T. Andernach, H. Kloosterman, R. Bos, and A. Nijholt. Building dialogue systems that sell. In *Proceedings Natural Language Processing and Industrial Applications*, pages 41–46, New Brunswick, Canada, June 1996. 1, 1
- [WP85] David L. Waltz and Jordan B. Pollack. Massively parallel parsing: a strongly interactive model of natural language interpretation. *Cognitive Science*, 9(1):51–74, 1985. 38
- [Yas84] Hiroto Yasuura. On parallel computational complexity of unification. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'84)*, pages 235–243, Amsterdam, 1984. Institute for New Generation Computer Technology [ICOT]. 7

- [YO94] Akinori Yonezawa and Ichiro Ohsawa. Object-oriented parallel parsing for context-free grammars. In Adriaens and Hahn [AH94]. [33](#), [34](#), [35](#), [36](#), [36](#), [36](#)

