CWI Tracts

Managing Editors

J.W. de Bakker (CWI, Amsterdam)
M. Hazewinkel (CWI, Amsterdam)
J.K. Lenstra (CWI, Amsterdam)

Editorial Board

W. Albers (Maastricht)
P.C. Baayen (Amsterdam)
R.T. Boute (Nijmegen)
E.M. de Jager (Amsterdam)
M.A. Kaashoek (Amsterdam)
M.S. Keane (Delft)
J.P.C. Kleijnen (Tilburg)
H. Kwakernaak (Enschede)
J. van Leeuwen (Utrecht)
P.W.H. Lemmens (Utrecht)
M. van der Put (Groningen)
M. Rem (Eindhoven)
A.H.G. Rinnooy Kan (Rotterdam)
M.N. Spijker (Leiden)

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The CWI is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

CWI Tract 55

A program generator for recognition, parsing and transduction with syntactic patterns

G.J. van der Steen

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

1987 CR Categories: F.1.1, F.1.2, F.2.2, F.4.2, H.3.1, H.3.3, I.2.2, I.2.7, I.5.4, I.7, J.5.
ISBN 90 6196 361 3
NUGI-code: 811

Copyright © 1988, Stichting Mathematisch Centrum, Amsterdam
Printed in the Netherlands

Preface and acknowledgements

This study is an account of an evolutionary process. It describes the development of a program-generator during the years that I worked in the computer department of the Faculty of Arts of the University of Amsterdam. There I was confronted with a number of computational problems, which initially seemed to diverge widely, but which gradually became unified when I tried to develop formalisms for them. It was fascinating to extend the technique of LR parser-generation and the construction of formal automata for which programs can be generated. I suspect that a number of my research fellows in the faculty, who were my clients, initially suffered from my tendency towards unification. But I hope I gave back a system in which they can express far more problems than they ever expected, and with a runtime behaviour which excells similar systems. In general I thank the faculty for the confidence they had in the way I thought things had to be done. The resulting system has now found its way to the industry.

During my research I cooperated with a large number of people. Hugo Brandt Corstius specified the first wishes for a pattern recognition program for large corpora of texts. Jan van Katwijk generously provided me with the code of his own LALR(1) parser-generator, which served for me as a solid initial system. For the programming of the compiler part of the Parspat system I got the invaluable help of Marijke Elstrodt. We went through a number of crises during the time that I extended the formalism together with new techniques for compilation. Marijke had to keep the system running for all former applications and at the same time program the new techniques. The runtime part was partly programmed by Menno Ytsma, who mysteriously appeared out of nothing, volunteered unpaid for a year, and then disappeared when the stock market no longer obeyed his calculations. Casper Dik performed a good job by extending the runtime part and keeping it in a healthy condition. Pieter Masereeuw has taken up the development of user-interfaces. Joos Skolnik programmed the concept of the external trie-datastructure.

During a number of years the use of grammatical formalisms in Corpus-linguistics is debated in working groups of the ZWO-werkgemeenschap "Computerlinguistiek en Mathematische Linguistiek". I profited from discussions with Jan Aarts, Dik Bakker, Marcel Dekker, Hans van Halteren, Theo van den Heuvel, Job Honig and Jan de Jong. Suggestions for the improvement of programs I got from numerous people, but especially I like to thank Bieke van der Korst, Jurjen van den Hoek, Ph. Houwink ten Cate, Jan de Jong, Pieter Masereeuw, Willem Meijs, Gerard Molenaar, Leo Plenckers, Gerhard Riesthuis, Gerjan van Schaaik and Thera Wijsenbeek for their close collaboration and for their willingness to provide me with examples of the use of the Parspat system and its predecessors.

With the research department of BSO we set up a good collaboration, not least because of the excellent work being done there by Bieke van der Korst.

I thank Jan van Leeuwen for the careful reading of versions of my manuscript and for his suggestions for improvement. He has the gift to be critical and stimulating at the same time.

Wim Klooster I thank for his continuous interest in the application of Computer Science in the Humanities and for his comments on earlier drafts of this book.

Willem Meijs corrected and commented on my English.

Contents

1. Introduction and goals

Syntactic descriptions play an important role in numerous applications from Linguistics, Artificial Intelligence, Musicology, Biology, Shape Analysis. They are a model of the apparatus by which we can select meaningful impressions from the outside world and by which we can express our messages to that world.

Most of the formalisms for syntactic descriptions stem from Linguistics. According to these formalisms it is possible to generate, to recognize, to parse and to transduce parts of sentences of a language in a precise way. "Syntactic Pattern Recognition" (abbreviated "SPR") is, at this moment, an emerging field (Gonzalez and Thomason, 1978) (Fu, 1982) in which the syntactic methods of Linguistics are applied to other fields. A central role is played by the recognition and transduction of parts of datastructures of atomic symbols, formulated in some grammatical formalism.

The potential applications of SPR are numerous. All natural objects in which we can project some ordering and all artificial objects in which some kind of ordering or sequencing plays a role are candidates for syntactic description and thus become objects for a syntactic treatment.

Images of natural objects are perceived in a psychophysical way. It may not be obvious in which way we should segment the object. Pavlidis (1977) pointed out that the image segmentation problem is therefore not susceptible to a purely analytical solution. This gives rise to the problem of ambiguity.
Fu (1982) commented that any mathematical algorithm must be supplemented by heuristics, usually involving semantics, about the recognition of the class of objects under consideration.
In natural language applications the same may be true on the lexical level. It is emphatically true when we deal with spoken language. Only systems which are transparent for ambiguities arising from different possible ways of perception and interpretation have a chance to become instruments for semantic analysis.

In this book we consider the computational aspects of recognition, parsing and transduction.

There are a number of applications in Computational Linguistics which were frustrated and eventually stopped by the lack of efficient computational strategies. In such a growing field the computer programs should be of a sufficiently general and optimized nature to become a stimulus instead of a hindrance to new research-projects. The programming should preferably be done in an automatic way, straight from the formalism in which problems are expressed.

The attempts to automatically transform grammars into recognizing, parsing or transducing programs have their roots in Computer Science, where a rich literature originated on automatic parser generation for programming languages. These techniques can be extended to the domain of SPR.

In this book we provide a program generator for a formalism which unifies a number of popular grammatical formalisms which are in use in SPR.

In chapter 2 we will describe this unifying formalism. In chapters 3 and 4 we will describe a formal machine model (a "Parallel Transducing Automaton", abbreviated "PTA") for which programs can be generated from grammars which are written in the unifying formalism. A concrete implementation of the machine and the accompanying program generator are discussed in chapters 5 and 6. The name of this implementation is "Parspat" (a "Parser for Pattern Grammars"). We will investigate the complexity of the generated programs in chapter 7. Finally, in chapter 8, a number of applications are shown which are treated with the aid of Parspat.

In order to set the scene we proceed with a short introduction to the computational aspects of SPR, the use of grammatical formalisms and the construction of parser generators. We will indicate which problems will be treated, and we will end with a list of results achieved in this book.

In order to show the computational aspects of a computational grammatical system, we can view it as an information system. We can then make use of the traditional division between the functional, analytic and genetic aspects of such a system.
In the course of our discussion we will indicate in italics which problems will be treated and what will be developed. The discussion will be as global as possible, only presenting details in order to indicate what has been done already and what the current bottle-necks are. Of course we will not treat every problem in this book, but we will at least give an opinion on how future work can be done. In order to do so we present the three aspects in an evolutionary form.

The analytic aspect describes the internal working of a computational grammatical system as a simulation of a user-defined process.
The functional aspect describes the interaction of a computational grammatical system with its environment. We will discuss briefly the aspects of input and output and, if the system is a component of a larger computational system, the form of intermediate datastructures.
The genetic aspect describes the way in which computational grammatical systems originate and change in the course of time.

1.1 Analytic aspects of computational grammatical systems

The analytic aspect describes the simulation of a user-defined process. In computational grammatical systems the following processes play a role :

- recognition, which can be simulated by formal recognition
- interpretation, which can be simulated by formal parsing
- change, which can be simulated by formal transduction.

We give a short recapitulation of some characteristics of the three processes and indicate in which kind of applications they play a role.

1.1.1 Recapitulations

A *recognizer* is a process which takes as input a string and either accepts or rejects it depending on whether or not the string is a sentence which may be produced by the syntactic description.

A *parser* is a recognizer which also outputs a structural description of the sentence according to the syntactic description.
A transducer is a recognizer or parser which emits output-symbols.

A process is said to be :
- Off-line when a response about rejection or acceptation is given after processing of the whole input. This assumes that the output is of limited length.
- On-line when there is a response after each symbol of the input, indicating possible continuation or rejection.
- Real-time when the on-line response will come within a guaranteed time. The term indicates the cooperation with other processes where this guarantee is imperative; usually, rejection will not occur and the input is potentially of unlimited length.

In the following table we indicate for some typical applications the desirable characteristics of their implementation.

	recog- nition	par- sing	trans- duction	off- line	on- line	real- time
Typical applications :						
- eeg-analysis	x					x
- shape analysis		x			x	x
- pattern-invocations	x			x	x	
- text manipulation	x				x	
- exploration of texts		x				x
- free-text database systems	x				x	
- text -to-speech systems			x	x	x	x
- speech-recognition	x	x			x	x
- grammatical analysis		x	x		x	x
- machine translation		x	x	x	x	x

Current bottle-necks are : the limited availability of on-line and real-time recognition, parsing and transduction algorithms for a large number of syntactic formalisms.

In this book we concentrate ourselves on on-line recognition, parsing and transduction.

Ways of analysing complexity: exponential versus polynomial

We recapitulate here some familiar complexity classifications from (Garey and Johnson, 1979). The "time complexity function" for an algorithm expresses its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size. (The term "problem" may, without loss of generality, be interpreted as an algorithm for the recognition of some input.) A "polynomial time algorithm" is defined to be one whose time complexity function is $O(p(n))$ for some polynomial function p, where n denotes the input length. Any algorithm whose time complexity function cannot be so bounded is called an "exponential time algorithm". If $p(n)$ is a linear function in n we speak about a "linear time algorithm".

Many exponential time algorithms that we encounter are merely variations on exhaustive search, whereas polynomial time algorithms generally are made possible only through the gain of some deeper insight into the structure of a problem. There is a wide agreement that a problem has not been "well-solved" until a polynomial time algorithm is known for it.

Hence, we shall refer to a problem as *intractable* if it is so hard that no polynomial time algorithm can possibly solve it. Problems for which polynomial time algorithms exist are usually classified as "P-problems", in contrast to "NP-problems". For the latter ones no polynomial time algorithm has been found, but there is a polynomial time algorithm to verify that given solutions are correct. A problem is NP-complete if it is in NP and every NP-problem is polynomially reducible to it [1].

Error detection and -correction

Errors in a text (according to the grammar) have to be detected as soon as possible. Analogous to programming languages a further requirement may be that the error will be "repaired", as correctly as possible, in order to continue the recognition- or transducing-process. The strategy for this repair of "ill-formed input" depends heavily upon the parsing-strategy chosen.
On-line processes require that errors are reported as soon as possible.

In this book we will only give an advice for the building of an error detection and -correction component in the program generator which we will develop.

1.1.2 Sources of ambiguities

During recognition, parsing and transduction ambiguities may arise. The causes may be :
- the input contains variants;
- according to the syntactic formalism there are two or more possible ways to proceed in order to reach a final structural description;
- inherent ambiguity: according to the syntactic formalism there are several structural descriptions possible;
- built-in strategies for error-recovery which try to recover along different paths;
- built-in strategies for the handling of ill-formed input.

With online recognition of ambiguous grammars there are 3 possibilities of proceeding after a new symbol is read in:
1. no proceeding possible (and an error-recovery scheme may have to be invoked);
2. there is exactly one possibility of proceeding (then the process is said to be deterministic);
3. there are two or more possible ways of proceeding (the non-deterministic case).

Ambiguities may be resolved (Winograd, 1983, p. 368) by *Explicit backtracking, Chronological backtracking*, *Deterministic parsing* and *Parallel parsing*.
Explicit backtracking is provided in some interpreters of formalisms and in some programming languages (like Icon, Summer, Prolog etc.).
Chronological backtracking is the simple mechanism built into a number of formalisms and programming languages in order to try out all possibilities ("depth first").
Deterministic parsing forces one choice out of many. It is sometimes called "heuristic search". This may be interesting if one wants to test the hypothesis that the machine can be deterministic in the cases where a human being is deterministic too (Marcus, 1980).
Parallel parsing tries to follow all paths in parallel ("breadth first"). It is attractive for the parsing of context-free grammars, because of the possibility of cubic processing time,

[1] For details, see Garey and Johnson (1979)

caused by the sharing of common paths. Winograd doubts if more complex formalisms can have the same kind of efficient processing.

Bottleneck: in nearly all existing implementations of grammatical systems the handling of ambiguity is poor. Most implementations work with chronological backtracking techniques. In the on-line case this leads to exponential runtime behaviour.

In this book we commit ourselves to parallel parsing within polynomial time when this is theoretically possible. We will provide for a technique for "shallow" and "deep" binding of variables and output in shared parse trees.

1.1.3 Formalism, programming and program-generation

1.1.3.1 Formalism as static description of a process

For our purposes we define a formalism as a datastructure which can be interpreted. The semantics of the interpreter will determine which (input, output) pairs are valid (if such a determination is possible). Stated in a dynamic way an interpreter transforms an input into an output. The goal of the simulation is reached when the same (input, output) pairs are valid for both the human user and the machine.

We will now summarize how the programming of the machine is achieved. We will do this from the viewpoint of a linguist who wants to create a running process on a computer. In our discussion we follow an evolutionary trail.
For the linguist there are two possibilities : (1) writing a program (the programming language is, by default, the formalism) or (2) writing a grammar in a syntactic formalism.
For the machine there are also two possibilities : (A) executing a machine-program or (B) running an interpreter for a formal machine which executes a program generated for that formal machine.
Interacting between the linguist and the machine there may be a compiler which transforms a syntactic description into : (I) machine code for the hardware of the machine, (II) program code written in a programming language, or (III) machine code for the formal machine. (We leave out the usual compilers, assemblers and loaders which are necessary for the treatment of programs written in a programming language.)

The following combinations are currently practised :
-1- (1) (A) : the linguist writes a program directly in a programming language and runs it on the hardware. Early machine-translation systems were written directly in some programming language. We quote Winograd (1983): "That work predated much of the work on formal linguistics (and contributed to it), so from our current standpoint the computational techniques and theories of syntax look chaotic and outdated.. ". This is the oldest situation, and still in practice. There may be several reasons for it:
- the linguist is not able to write his syntactic description in one of the existing formalisms
- maybe he is able to do so, but there is no program generator available for that formalism
- there is a program generator available for the formalism, but it takes too much time in the development phase to work with it, or it lacks sophisticated debugging tools.

The disadvantages of this situation are numerous, but some of them are hidden for the starting linguist-programmer. They will manifest themselves gradually, but painfully, in the course of his evolution. We make the following observations :

- The choice of the programming language is important. Some languages lend themselves better for the programming of recognition and parsing processes than others. Languages like Snobol, Icon, Summer, Prolog have suitable built-in operators. Snobol has a wealth of pattern matching functions, Prolog facilitates the writing of attribute grammars. A linguist may be happy when he needs nothing more than the built-in functions. But if he wants to change his formalism he may be in trouble.
- General-purpose programming languages have no facilities for the handling of ambiguities. The special languages which we noted have built-in strategies for the handling of ambiguities which all rely on backtracking. The exponential nature of backtracking manifests itself when larger programs are to be run. A typical consequence are the ad-hoc solutions which are invented by the grammar writers in order to restrain the combinatorial explosions. However, with every change of the grammar these inventions have to be adjusted. A striking example of the resulting folklore is provided by Finin (1983, p. 17): "The grammar writer may find this a useful arc in that it gives him greater control over the automatic backtracking done by the Backhouse interpreter."
- In some applications the adjustment of the grammar will, in principle, never end. If the formalism is a programming language one has to be aware of the maintenance costs, which can become prohibitive.
- Sequencing. Translation of grammar rules by hand into a program-notation implies a mapping onto sequential operations, even if no sequence was intended.

Winograd (1983, p. 310) describes the result: "The parser in SHRLDU... Its grammar covered a relatively wide range of English phenomena, but it was complex and difficult to modify, since the interactions among rules were implicit in the order that things got done in the program."

Grammars for natural language are in a constant process of revision and enlargement. During this process the complexity of the formalism will collect its debts. It depends directly upon the number of relations between the grammar rules. Forced sequencing will considerably increase this number.

Some syntactic formalisms make use of explicit sequencing of rules, like the ordering of transformations in transformational grammars and the ordering of phonetic rules. It is a characteristic of general rewrite schemata that the order in which rules are applied may influence the result. The decision to order rules should therefore remain visible in the formalism, and not be mixed-up with the sequencing within a program.
- Only a few linguists will be able to read a large computer-program. For all others the program can only be judged by its behaviour, and its theoretical value will be nil. Sometimes the runtime behaviour of a program is heralded as being of theoretical value. In that respect psycho-linguistic arguments have been used, for instance to justify the sequencing of programming statements. We argue that these arguments should be stated beforehand and should play a role throughout the whole design process.

-2- (2) (1) (A) the linguist writes a grammar and transforms it manually, in a systematic way (as in "recursive descent" methods), into a program in a programming language and runs it on the hardware. This method removes some of the disadvantages which we noted above. But all arguments against the use of a programming language remain the same.

-3- (2) (I) (A) the linguist writes a grammar which is compiled into machine code for the hardware of the machine and runs it on the hardware. This is the method which is employed in parser generators for grammars which describe programming languages. The resulting parsers are intended to run deterministically. Well-known are the LL- and LR-parser generation techniques. These generate code for a pushdown automaton (PDA). Most computer-processors have hardware (stack-)instructions for the handling of a PDA and therefore these parser generators are able to generate programs which are stated directly in these instructions. The resulting parsers are therefore fast. Applications which make use of non-ambiguous context-free grammars can make use of such parser generators.

-4- (2) (II) (A) the linguist writes a grammar which is compiled into program code written in a programming language and runs it on the hardware. The same technique is used as in -3-, but now the resulting program is written in a programming language. The advantages are that the parser generator becomes more portable and that built-in functions can be used, like i/o statements. A disadvantage is the decrease in speed. Well-known examples are the utilities Yacc and Lex in Unix. The allowed formalism is the same as in -3-, but more recently parser generators for attribute-grammars are being developed as well.

-5- (2) (III) (B) the linguist writes a grammar which is compiled into machine code for a formal machine and runs it on the interpreter for the formal machine. The parser-generation methods of -3- and -4- were successful because they generate in a clean way code for a formal automaton, in this case a PDA. This automaton is directly related to the formalism of a context-free grammar. Recognition and parsing of ambiguous context-free grammars with the aid of a PDA requires the technique of backtracking. In order to process ambiguities in polynomial time ánd to process more evolved syntactic formalisms it is necessary to develop a more complex automaton.
Because there is, up till now, no hardware implementation of the instructions of such automata, interpreters for these automata are necessary.

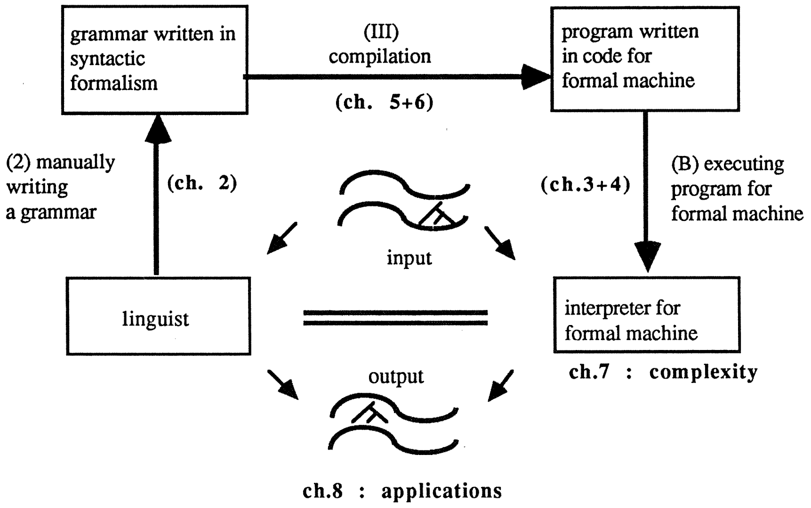
In this book we will develop the "Parallel Transduction Automaton" (PTA) for situation 5.

More advantages of program-generation are :
- restoration of the central use of the formalism for the *description* of language, as opposed to the *recognition* or *generation* of language. It leaves room for the automatic treatment of ambiguity;
- no programming efforts;
- possibility of greater efficiency of the resulting programs; research on improvement can be separated from the applications;
- there is no temptation for the linguist to change the grammar over and over again in order to increase its efficiency;
- if sequence is specified explicitly it becomes possible for a parser generator to combine parallel rules in order to optimize the recognition-process.

In the foregoing we mentioned one reason why a linguist, despite the availability of a program generator, will stick to the writing of his own programs : it takes too much time in the development phase to work with it, or it lacks sophisticated debugging tools.

In the interpreter for our PTA we incorporate tools for debugging.

In this book we opt for strategy 5. Its organisation ((2)(III)(B)) is depicted in the following figure. We indicate the chapters in which the different parts will be treated.


grammar written in syntactic formalism
(III) compilation
(ch. 5+6)
program written in code for formal machine
(2) manually writing a grammar
(ch. 2)
input
(ch.3+4)
(B) executing program for formal machine
linguist
interpreter for formal machine
ch.7 : complexity
output
ch.8 : applications

1.1.3.2 LR parser-generation and beyond

In general a LR parser generator tries to construct a finite automaton with states associated with all possible combinations of positions in grammar rules where a parse could be, at the same time. Because of recursion in the grammar, already existing states may be generated again; in that case identical states are merged and the distinction between the two states, which depends on the history of recognition during runtime, is postponed to the run-time stack. In addition to the creation of a finite automaton instructions can be generated for the run-time stack. Non-determinism is introduced when on the reading of an input symbol a transition can be made in the finite automaton as well as an instruction can be performed for the stack. In that case one or more look-ahead symbols (up to a certain maximum k, determined during compile-time) have to be sufficient to make the process deterministic (if this is not the case the grammar is said to be not LR(k)).
LR-grammars are unambiguous. They are a subset of the context-free grammars and are mainly used for the description of the syntax of programming languages.
It is possible to generate with the same LR-technique code for a PDA which has a dag-structured stack. Tomita (1986) first reported on this. Independent of him we developed the same method but extended it for a number of generalizations of the context-free formalism.

In this book we extend the LR(0) parser-generation technique for the treatment of Chomsky type-0 grammars and transduction grammars, both extended with regular expressions, don't cares, tree symbols, Boolean operators and variables.

1.2 Functional aspects of computational grammatical systems

The functional aspect of a computational grammatical system describes the interaction with its environment. We distinguish four components :

1. input from the environment to a grammatical system,
2. output from a grammatical system to the environment,
3. the interaction between components of a grammatical system,
4. lexicons.

1.2.1 Characteristics of input

Input to a recognition- or transducing-process is text. In its broadest definition it is a sequence of discrete entities. These entities may stem from written characters, music, speech, signals. The text may be already organized in some data-structure, or, on the other hand, it may have to be captured from an uninterrupted stream of information, hidden in a signal. Nearly all texts contain errors, originating from human interfering or from physical distortions.
Texts are often embedded in some medium which itself may be a text too. These "carriers" have to be removed before the actual treatment of the text starts. We may here think of texts originating from photo-typesetters, transcriptions of dialogues or simply a corpus with a wealth of enrichments, from which we only want to use the "bare" text.
Texts may contain "don't cares" which can get a (multiple) assignment during the analysis of the text.
Alternatives in a written text may arise from alternative readings in manuscripts. For an example we refer to section 8.1.When a text contains alternatives it may take the form of a (labeled) network. When texts are enriched by codings on the text-, word- and higher levels alternatives may arise from unresolved ambiguities.

In this book we permit the input to the PTA to be in the form of a labeled tree.

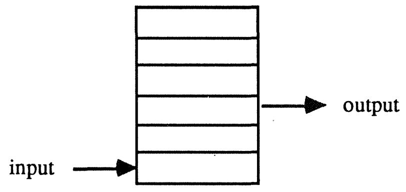
1.2.2 Characteristics of output

A recognition process outputs, in principle, "yes" or "no" .
A parsing process outputs a structural description of the sentence according to the syntactic description. In the case of a context-free grammar this is a parse tree.
A transduction process emits output-symbols. We distinguish two situations. The first one concerns a recognition process which generates a message when some point in the (pattern-) grammar is reached. We call such messages "reports". The purpose of these symbols is to trigger other cooperating processes, written in some programming language. These processes are often embedded in an on-line environment.
The second situation in emitting output-symbols is to transform a text. In the case of "pattern-transduction" pieces of a text will be transformed.

The PTA output may contain: structural descriptions, reports, the variables which are associated with the start-symbol and transformed text. The output will be generated as early as possible.

1.2.3 Characteristics of the interaction between components of a grammatical system

The output of a transducer may be the input to another one. If the transducers are produced according to a grammatical formalism we may speak of "cascaded grammars". For instance this may be the case in general transduction schemes for spoken language or for machine translation. The sequence in the cascade has to be explicitly defined.
In the following figure we depict the layers of a grammatical system, but abstract away from its correspondence with linguistic levels. We suppose that the process for each layer is constructed according to a grammar. The whole grammatical system consists of these cascaded grammars. Such a grammatical system may be used for different purposes. It is possible that for some of these purposes some layers do not have to function. Input and output of the whole system are, in general, not related to the first and the last layer. In our discussion these aspects will not be of relevance.



recognition in a layered system

Two situations can be distinguished. In the first one there is no feedback between the layers. In the second situation feedback between a number of layers is necessary. We give some examples.

1.2.3.1 First situation : no feedback between layers

It is possible that in a cascade of transducers, consisting of e.g. two transducers, the first one is of a fixed nature, but the second one is in an experimental stage. One might think of the first one as transducing raw input text into a well-defined datastructure, e.g. a tree, with each node enriched ("decorated") with information from a lexicon. The second transducer (the last one in a cascade can also be a parser, or a recognizer) is doing grammatical analysis according to an ever changing (and hopefully improving) grammar. Input to this transducer is then the decorated tree. This tree has to be formed once and stored in external memory : it is an intermediate text.
One may imagine a second step: after the settlement of the second grammar correct parse trees are created, as structured output. These parse trees may be stored as intermediate texts (including ambiguities). Then a third recognizer may be constructed, acting as a pattern matcher, in order to search for and to count syntactically interesting patterns.
This example describes current practice in Computational Linguistics (cf. Aarts and Van den Heuvel, 1984).
Suppose that even the third (pattern-)grammar becomes a fixed one. Then one may imagine that all three transducers work together as co-processes, transmitting output as soon as possible to each other.
One step further could be that all three transducers are combined into one and that an automatic parser generator creates an overall, efficient, recognizer which is able to locate in raw

texts, stemming from photocomposers, meaningful syntactic patterns, expressed in elements of parse trees.
In this evolutionary story we may first start with off-line processes, but in the last stage on-line processes become a necessity.

1.2.3.2 Second situation : feedback between layers

We give four examples. They stem from the applications of Grammatical Analysis for document conversion, Machine Translation, Automatic word- and sentence coding in Corpus Linguistics and Speech Recognition. In the first application the layers correspond with the context-free parsing of a sentence and the recognition of restriction-patterns in the emerging parse trees. These patterns serve to restrict the generation of ambiguous parses which are not allowed. In the other applications the layers correspond with two or more of the traditional linguistic levels of phonological, morphological, lexical, syntactic, semantic, pragmatic and discourse analysis.
The four applications have in common that they (want to) make use of integrated linguistic grammatical knowledge on a number of levels together, but that each level is implemented according to the insights of specialists for each of the levels. In all four applications there is an urgent need for a simple and efficient feedback between the levels.

1.2.3.2.1 Text analysis: grammatical analysis for document conversion

Winograd (1983) describes 50 working systems which are based on some grammatical formalism. The most outstanding program in the category "text analysis" is the "Linguistic String Parser" of the group of Sager (1981) in New York (see also section 2.3.1.1). We quote partially from a summary in (Hobbs and Grishman, 1976) : "It is capable of handling a great proportion of English grammatical constructions, including conjunctions and comparatives. It contains a two-stage syntactic analysis of English sentences into a simple underlying representation. The output of the first stage is a set of trees making explicit the surface structure of an English sentence; each tree represents a syntactically valid analysis of the sentence in accordance with the linguistic string theory of Harris (1962). The second stage takes these parse trees and transforms them into a kind of predicate notation.
The grammar for the first stage consists of a set of (about 180) BNF productions and a set of (about 180) conditions on the application of these productions; the conditions are expressed in a specially designed Restriction Language. The grammar for the second stage consists of transformations written in an extension of the Restriction Language. Each transformation performs certain tests on the structure of the tree and the attributes of the sentence words, and, if the requisite conditions are met, alters the tree. The sequencing among the transformations is specified entirely within the transformations themselves" (end of partial quotation).
The system is in use for the automatic transduction of medical reports into entries for a database system. It uses a restricted lexicon, specifically designed for that application. See for an example section 2.3.1.1 .

1.2.3.2.2 Machine Translation

In (Maas and Maegaard, 1984), the Eurotra-formalism (as developed up till that moment) is described for writing transduction processes which operate on trees. These processes consist of sub-processes, operating in parallel or serial and each consisting of a set of general rewriting rules. The rules are unordered. Each process has its own allowed input- ("expectation") and output- ("goal") datastructure, which serve as filters. According to the

documentation, the treatment of ambiguities has to be completely transparant for the writer of the grammar rules.
In (Slocum, 1985) a survey is presented of current machine translation projects. The METAL German-English system accommodates a variety of linguistic theories/strategies. The German analysis component is based on a context-free phrase-structure grammar, augmented by procedures with facilities for, among other things, arbitrary transformations. The English analysis component, on the other hand, employs a modified GPSG approach and makes no use of transformations.
The Ariane-78 translation system (developed in Grenoble by the GETA institute) supports grammars which are actually networks of subgrammars; that is, a grammar is a graph specifying alternative sequences of the applications of the subgrammars and optional choices of which subgrammars are to be applied. In principle this system is open ended and could accommodate arbitrary semantic processing. It nevertheless suffers from a rigid implementation in low level languages. We quote: "As a result, the GETA group has been unable to experiment with any radically new computational strategies. Back-up, for example, is a known problem (Tsjujii, personal communication): if the GETA system 'pursues a wrong path' through the control graph of subgrammars, it can undo some of its work by backing up past whole graphs, discarding the results produced by entire subgrammars; but within a subgrammar, there is no possibility of backing up and reversing the effects of individual rule applications" (end of quote). In fact, the formalism of Ariane-78 and the proposed Eurotra-formalism resemble each other.
About the multilingual system SUSY (developed in Saarbrucken): ".. the linguistic rules were organized into strictly independent strata and, where efficiency seemed to dictate, incorporated into the software. As a consequence, the rules were virtually unreadable, and their interactions, eventually, became almost impossible to manage".
Slocum concludes that "it is critically important that a development group be able to conduct experiments that produce results in a reasonable amount of time. After too long a delay, the difference becomes one of category rather then degree, and progress is substantially - perhaps fatally - impeded."

1.2.3.2.3 Corpus Linguistics

A broad definition of a corpus may be: a fixed amount of text, selected and prepared for some analysis. Sources of texts may vary widely. They may be transcribed from old manuscripts, from spoken language (for instance for the study of dialectal variations or of children's language), or they may originate from wordprocessors, photo-composing machines and automatic reading machines which process journals or dictionaries. Sometimes it is necessary to discriminate scrupulously in order to prepare statistically relevant portions of texts. On the other hand linguists may use a corpus in order to browse through them in search of some interesting pattern, unforeseen by the original compilers of the corpus.

There are several computing centers in the Humanities all over the world which act as archives for machine readable texts (e.g. in Oxford, Pisa, Irvine). A number of corpora are available for general use which are provided with tags at word-level. As examples we mention the following corpora :
- The Brown Corpus, American English, 1 million words, 16 genres, morphologically coded (Francis and Kucera, 1964). Results of research based on this corpus are frequently published in (Icame-News). Some publications are (Hofland and Johansson, 1982), (Meijs, 1982), (Meijs, 1984), (Francis and Kucera, 1982), (Leech, Garside and Atwell, 1983b).

- The Lob Corpus, British English. Analogous to the Brown Corpus (Johansson, Leech and Goodluck, 1978). Results again in (Icame). Further: (Atwell, Leech and Garside, 1984), (Hofland and Johansson, 1982).
- The "Eindhoven Corpus", (600.000 words, 9 genres), Dutch language, morphologically coded (Uit den Boogaart, 1975). Results e.g. in (Renkema, 1981).
- The "Liege Corpus" (300.000 words), Latin, with morphological codes and lemmata. Results e.g. in (De Jong and Masereeuw, 1985).

Corpora are used for obtaining quantitative measurements on the use of language. These measurements may be of interest for a number of purposes. Among them we name Corpus Linguistics and applications in the "Language Industry".

In Corpus Linguistics corpora are the "physical" objects on which linguistic hypotheses have to be validated. The hypotheses may concern the applicability of grammar rules and restrictions on the use of them, or the characterization of different groups of language users, different registers, etcetera. From recent literature we quote a few topics :
- differences in style (Renkema, 1981),
- collocational distinctiveness: "All recurring sequences are potential collocations" (Kjellmer, 1984),
- adverbial realisation and position (Quirk, 1984),
- elliptic structures and their syntactic description (Meijs, 1984), and
- postmodifying clauses (De Haan, 1984).

Atwell, Leech and Garside (1984) describe two potential industrial applications of corpus-based research:
- in IKBS (Intelligent Knowledge-Based Systems) applications, such as testing and improvement of stylistic acceptability of texts produced by natural language synthesis programs: acceptability of style and collocations,
- application to word-processing, especially when probabilistic parameters are included.

Resources for the applications will be grammars and dictionaries of a probabilistic nature, derived from corpus-based research. At the phrase level there are proposals for the construction of a "distributional lexicon" (Atwell, Leech and Garside [1984]) which has to contain:
- significant word-word collocations and combinations,
- significant combinations, positive or negative, of words and syntactic constituents containing them, and
- significant combinations of a word and an adjacent syntactic category.

Both kinds of applications have to make use of quantitative measurements. These may concern prosodic elements, syllables, phonemes, combinations of characters, morphemes, wordforms, lemmata of words, combinations of words, part-of-speech codings, grammatical constructions and combinations of elements from these various domains (Engels [1981], Francis and Kucera [1982], Geens, Engels and Martin [1975], Hofland and Johansson [1982], Renkema [1981], Uit den Boogaard [1975]).

In order to obtain quantitative measurements and to facilitate grammatical analysis it is often necessary that corpora be enriched with codes (or "tags"). These codes may be attached on different levels (Eeg-Olofsson and Svartvik, 1984), such as text, word, phrase, clause and discourse level. Tagging on a higher level usually implies tagging on a lower level. The tagging may be done by hand or (mainly) by a computer program.

On the text-level we may expect codings which depend on the sources of the material. Transcribed spoken language may be provided with prosodic markers (stress, pitch indications for e.g. the fall or rise of the voice, stress, pauses, duration) and tone unit boundaries.
Transliterated text and old manuscripts may be enriched by philological marks such as: "broken", "difficult to read", "possible reading is ..", "variant of .." etcetera (see also section 8.1). In some cases the tagging of a corpus on the word level is used mainly as a stepping-stone to phrase level tagging. In that case the system of word-class tags will continually be revised and supplemented to suit the needs of higher-level tagging (Eeg-Olofsson and Svartvik, 1984).
The composition of these tags varies from corpus to corpus, depending on the initial purposes for which a corpus was coded. Usually they are constructed hierarchically, consisting of characters and digits. These tags may be of a morphological, grammatical or semantic nature.
It is not a trivial task to define units on the word level. Among the problem-cases are: contractions ("don't"), multi-word adverbs ("as well"), conjunctions ("as though"), pronouns ("each other"), complex prepositions ("from the point of view of"), complex adjectives ("up to date") and proper names ("New Guinea").
We quote (Eeg-Olofsson and Svartvik 1984, p. 56) : "Multi-word tags cause certain technical problems of ambiguous segmentation. For instance, the number of tags assigned to the word combination "sort of" must be two in the context "that sort of life", but only one in "they sort of agreed" (end quote)".
There are fewer corpora with codings on the phrase level than with codings on the word level. The composition of the tags may vary with the grammatical insights of the composer of the corpus.
Some wordgroups, belonging to each other, may be separated by other wordgroups. This may be indicated by separate tags on the clause-level. Tags on the clause-level dominate tags on the phrase-level. They may signal the applicability of a transformational rule.
On the discourse level speech-specific items are handled that cannot be appropriately taken care of at the syntactic levels, like 'apologies', 'softeners' and 'hedges' (Stenström, 1984).

It is a major undertaking to shift the burden of coding from the linguist to the computer. It requires the translation of the skills of a linguist to a procedure which has to be implemented in a computer-program. These skills are often of an implicit nature, sometimes depending on years of experience. They may vary from linguist to linguist. One of the obvious benefits of such an undertaking, however, will be that linguistic insight is made explicit in a precise way.
Up till now linguistic scholars made their insights explicit in dictionaries and in grammars. The exploitation of these two tools had to be exercised and shortcomings had to be adjusted by intuition, especially when semantic features were needed.
This is the reason why the shift to machine-coding takes place in a gradual way. In an iterative way lexicons, grammar rules and programs are constantly being improved until they behave in a consistent and stable way for a fixed corpus. They may then be used on a new corpus which must be constructed in the same way as the former one. Implicit in this approach is the hypothesis that on a new corpus stability will be reached in less time. The testing of this hypothesis has to wait for more experience.
The process of automatic tagging on the word level may include the following stages (Garside and Leech [1982], Leech, Garside and Atwell [1983a, b], Atwell, Leech and Garside [1984]) :

1. pre-editing, to deal with formulae, capitalization and other oddities (may disappear when a precise formalism for the input is given),
2. programs to assign a list of one or more tags, in sequence of probability,

3. human post-editing, with a logging of detected errors and
4. proofreading of the enriched corpus.

Resources for the programs in 2. are:

A. procedures for doing morphological analysis of the word at hand. For languages with simple flexion these procedures may suffice with an inspection of a list of suffixes.

B. procedures for doing a dictionary-lookup. This dictionary may be an existing one or a list that is gradually built up during an initial interactive tagging.

C. procedures for doing some kind of syntactic "look-around". Up till now two methods have been used for this:

- "context-frame rules" : describing the local context of a word in order to disambiguate between several possible tags (Garside and Leech, 1982). These rules may be ranked by probability.
- a transition probability matrix (tunable for different genres of a corpus) for pairs or triplets of word-tags, again to be used in order to disambiguate.

The ultimate goal of automatic coding is, as Eeg-Olofsson and Svartvik (1984) point out: "to create a truly integrated system where knowledge derived from any level of analysis (e.g. syntactic expectations from the phrase level analysis) could be used to direct the tagging at any other level (e.g. the word level)".
The achievement of this goal will depend directly on the adaptability of programs to deal with grammars and lexicons of different kinds and on all levels. In contrast to the development and analysis of programming languages it is not possible to assign tags on a lower level without knowledge (explicit or implicit) of tags on a higher level.

There are a number of reasons why Corpus Linguistics is of computational interest.
Firstly, all aspects of Computational Linguistics are represented.
Secondly, often a transduction is necessary from the organisation of the text in manuscript form to an organisation suitable for the further automatic processing of the text. Sometimes the organisation of the text is not known and some discovery procedure has to be set up. In (Van der Steen, 1982) we described a general method for the syntactic description of the organisation of texts using extended context-free grammars in order to unravel the original text in atomic parts and to reorganize these parts (see also section 8.2).
Thirdly, the location of interesting patterns within a large corpus requires efficient recognition strategies. In (Van der Steen, 1982) and (Van Halteren, 1985) programs are described which assist the linguist in the formulation of queries in text corpora without the need to concern himself with a programming language. The command language of these programs acts like the query-language for a database system. The corpus, in this case, serves as the database.
Fourthly, the tagging and/or parsing of large amounts of texts demands the most efficient parsing strategies.
Fifthly, the amount of interaction between the different levels, which may correspond to the layers in a computational grammatical system.

The algorithms which are developed in this book are in daily use in the practice of Corpus Linguistics.

1.2.3.2.4 Recognition of spoken language

As we mentioned in the introduction, the segmentation of a continuous signal into discrete entities enables a syntactic approach. If the signal is speech then the segmentation may need all linguistic levels which we mentioned above.
On each level ambiguities may arise which may be resolved by analyses on other levels. Current systems implement this resolution in an ad-hoc fashion.
De Mori (1983) describes syntactic approaches to the interpretation of speech-patterns. Syntactic rules are mainly written in the atn-formalism or in markov-chains. The syntactic analysis of a speech segment is accompanied by probability calculations in order to prune unlikely continuations.
What is needed is a simple communication between the different levels which have to be processed. The ultimate goal is the same as was mentioned under "Corpus Linguistics".

In this book we will not handle probabilities within formalisms or texts.

1.2.3.3 Conclusions

From the preceding four applications we conclude that input and output of the whole system may occur in different layers. The layer in which the output occurs does not need to be the highest layer which is in function (for instance in the case of a speech recognition system where semantic analysis may be necessary in order to give a correct written representation of the spoken input).
With regard to the communication between layers we learn from the first application, text analysis, that the process of pruning of the results of a former layer works well. In that application the context-free parser generates a large number of parse trees which are subsequently checked by the layer with the restriction-rules. The feedback of the restriction-level can be of an on-line nature : it says yes or no to the possible continuation of the construction of a parse tree in the parsing layer. The message yes/no could be widened to the set of allowed next characters in the construction of the current parse tree.
From the other applications we learn that a closer cooperation between the layers is needed. The conceptual hierarchy of the layers does not imply a hierarchy in processing of the layers. In principle the processes in the layers run in parallel. Some experimental systems therefore work with a "blackboard" on which all layers may write and read messages. We interpret this "blackboard" as a set of global variables for all (or some) layers.

It is generally agreed that the use of global variables is an unsafe solution for the problem of communication between modules. They admit side effects, for instance in our case when (the all-pervading) ambiguities between layers arise. The usual remedy of procedure parameters seems to fail because of the parallellism of the processes. On the other hand, the layers form a conceptual hierarchy. From the viewpoint of the grammar writer who writes a grammar for some layer it is comfortable to have at his disposal the formalism of parameter passing to a lower layer.

Therefore, instead of global variables, a better idea seems to be to use variables as in attribute grammars and in Prolog, attached to nonterminals. In order to enable processing with incomplete results these variables must have the unification property, as in Prolog. For the grammar writer it is then transparant in which direction analysis and transduction works. For the processing it means that after the evaluation of a variable in one layer the evaluation of another variable in another layer can be completed.

It is desirable that in all layers the same grammatical formalism can be used. This formalism has to be sufficiently rich to support grammars for different layers. The interpreter of the formalism has to support unification for at least the variables that take care of the communication between the layers.

1.2.4 Characteristics of a lexicon

A lexicon attempts to describe the objects of a natural language, sometimes based on the intuition of the lexicon-maker, sometimes based on a large number of short quotations from the object-language. Objects on the word level are described by phonological, morphological, syntactic and semantic entities, sometimes with a circularity in the semantic description. In general the trend in linguistics is to increase the role of the lexicon and to decrease the role of the grammar.
Objects may have more than one description. They may be disambiguated in a more or less informal way by giving examples. Disambiguation in the lexicon may be more formalised by the introduction of a sentence-pattern, allocated to each alternative. We give an example from the Longman Dictionary of Contemporary English (LDOCE, 1978).
An example :

> The grammatical information to the verb "appear" is:
> appear v; wv6; I0, 3; (it) L (to be) 1, 7, 9; it + I5a, b, 6a.
> Here "v" indicates that we are dealing with a verb; wv6 that the use of it in the "progressive form" is infrequent; I0 that it may be used in an intransitive way; 3 that a "to"-infinitive construction may follow; L indicates that it may also appear as a copula (possibly preceded by "it" and followed by "to be") followed by a nominal (1), adjectival (7) or adverbial (9) constituent. I5 means that "appear" may be followed by a "that"-subordinate clause (in which case "it" has to be subject), where the "a" indicates that the word "that" may be omitted; "5b" that "so" or "not" may follow ("it appears so", "it appears not") as a replacement of a "that"-subordinate clause. "6a" indicates that a "wh"-subordinate clause may follow too.

This kind of information may be used in syntactic disambiguating strategies (Akkerman, Masereeuw and Meijs, 1985).

Conceptually, the lexicon is part of a syntactic description, but in practice it plays a separate role. Most implementations make use of an external datastructure for which it is necessary to know the whole entry before it can be looked up. This causes troubles with multiwords and words which appear in the grammar itself.

In this book we will restrict ourselves to the use of wordclass-information. We will use as an efficient external datastructure the trie to implement the lexicon. The access to the lexicon runs in parallel with the parsing process and is transparent for ambiguities, multiwords and words which appear in the grammar. The access-time is independent of the size of the lexicon.

1.3 Genetic aspects of computational grammatical systems

In the above we treated computational grammatical systems as information systems. In the case of the treatment of natural language these systems will never reach a final stage. There are two reasons :

- there is no natural language for which a complete grammar exists,
- there is a constant movement in linguistic theories, and change in formalisms.

We argued above that we can help the testing of grammars by the development of tools. One tool is a program generator.
Program generation for some formalism is in itself subject to a process of evolution. The typical stages are :
- the resolution of undecidability problems (usually with heuristics stemming from the application),
- the development of naive algorithms (usually exponential in running time),
- the improvement in speed to polynomial or linear algorithms,
- the improvement of the constant factors,
- the optimizations within the program, not dealing with the algorithm, and
- the adaptations in the hardware of the machine (e.g. development of micro code).

In this book we show, among other things,
- how to parse on-line in exponential time ambiguous type-0 grammars and transduction grammars, enriched with variables, with an on-line construction of the parses, with the aid of the "Parallel transducing Automaton" (PTA)
- how to parse on-line in polynomial time, with an on-line construction of the parse trees, extended context-free grammars with don't cares
- how to reformulate the problem of pattern matching of Aho and Corasick into the problem of LR-parsing, with the same linear time behaviour
- how to extend the LR parser generation method for the treatment of grammars with Boolean expressions
- how to extend the LR parser generation method for the treatment of term-rewriting systems
- how to improve the LR parser generation method in such a way that pattern matching with a number of keywords, containing regular expressions and don't cares can be implemented to run in sub-linear time
- how to generate in all these cases the minimum of code for the PTA in order to use only those parts which are necessary
- how to implement the PTA on parallel hardware.

The adjustment of a program generator to new formalisms can be established by :
- a description of the formalism by a (meta)grammar; notational variants between formalisms can be treated by changing this metagrammar or/and doing some preprocessing,
- incorporating existing formalisms as much as possible,
- trying to unify a new formalism with the already existing ones, and extending the program-generation technique.

With the transduction formalism developed in this book it is easy to write such preprocessors.

In this way we hope to contribute to the evolution of computational grammatical systems.

2. A unifying formalism

2.1 Introduction

What is desirable in a formalism for the automatic processing of texts ?
In a paper (1985) with the title "Hiding complexity from the Casual Writer of Parsers" Dahl writes as follows: "[It] raises the question of whether it is indeed possible to construct a formalism that combines efficiency with high expressive power, and that hides from the user all details that can be automated, thus providing a simple way of describing the purely creative grammar-writing aspects. So far it is difficult to see just how much should be made transparent, just how much expressive flexibility is appropriate without the formalism becoming too powerful and introducing new problems on this account, and just how to ensure efficiency without burdening the user with machine-oriented concerns such as control".

These remarks reflect the current state of the genetic aspect of computational grammatical systems which we discussed at the end of chapter 1. We suggested that high expressive power can be reached by unification of frequently used formalisms and that program generation should hide the details that can be automated. Program generation should be possible for all formalisms, however powerful they may be.

Winograd (1983) discusses a large number of syntactic formalisms. Some of them have been used only experimentally, others are in use in practical natural language systems. We are interested in the latter category and in the question how we can build effective systems.

In this chapter we present a unifying syntactic formalism and relate it to the formalisms which are in use in practical systems and in some applications ın SPR. One may change the syntax of the unifying formalism, to make it more readable or to adapt it to personal preferences. (At this stage we are not interested in notational variants of the formalism.) Then we will discuss the useful effects of the unification of existing formalisms. Finally we will summarize what has been done already on automatic program generation for formalisms which are embedded in the unifying formalism.

Terminology

In the sequel we will use the following abbreviations:

cf	for	context-free
cfg	for	context-free grammar
ecfg	for	extended context-free grammar
cs	for	context-sensitive
csg	for	context-sensitive grammar
lhs	for	left-hand side (of a rule)
rhs	for	right-hand side (of a rule)
sub-formalism	for	formalism that is embedded in the unifying formalism
FSA	for	Finite State Automaton
PDA	for	Push Down Automaton

Notations and definitions :
- a "symbol" is an atomic entity, such as a letter or a digit; in Parspat it is an Ascii-character
- an alphabet is a finite set of symbols
- a string (or word) is a finite sequence of symbols from some alphabet
- ε denotes the empty word
- A* is the set of all strings over the fixed alphabet A
- $\emptyset$ denotes the empty set

2.2 The unifying formalism

A grammar which is written according to the unifying formalism is called a "U-grammar".

Definition.
A U-grammar G is a 9-tuple (N, I, T, S, Z, C, R, P, M) where
- N is a finite set of nonterminal symbols; those symbols that are rewritten, as only symbol, at a lhs
- I is a finite set of intermediate symbols; intermediate symbols are introduced in a lhs of a grammar rule, where $|lhs| > 1$
- T is a set of terminal symbols (not necessarily finite); terminal symbols may be read from the input
- N, I and T are disjoint sets; V is the union of N, I and T
- S is a distinguished start-symbol, with $S \in N$ or $S = \varepsilon$; if $S = \varepsilon$ then the grammar is a transduction grammar, otherwise it is a Chomsky-type grammar
- Z is a finite set of variable symbols which can act as a parameter to a nonterminal symbol; expressions of variables and constants may be assigned to a variable and Boolean expressions can be formed with them
- C is a finite set of cooperation symbols with which Boolean relations between rules can be expressed
- R is a finite set of report-numbers with which a position within a rhs can be marked
- P is a finite set of rewriting rules
- M is an ecfg which describes the syntax of the symbols and the rules; we refer to this syntax by the term "unifying formalism"; M is called the metagrammar of G.

Δ

In the sequel we will use the following notations :
- a, b, c, ... (other than ε) are elements of T
- A, B, C, ... are elements of N
- w, x, y, ... are words in T*
- X, Y, ... are elements of V
- α, β, γ, ... are words in V*
- $|\alpha|$ denotes the length of α
- $j{:}\alpha$ is the j-th symbol of α if $1 <= j <= |\alpha|$.

Δ

For a U-grammar G= (N, I, T, S, Z, C, R, P, M) only the set P of production-rules has to be specified by the grammar writer. Usually the metagrammar M has been fixed for a number of applications. The other sets are derived from P with the aid of M in the following way.
The first rule of P determines whether G is a phrase structure grammar ("PSG") or a transduction grammar ("TDG"). If the length of the lhs of the first rule is 1 then it is a PSG and the notion at the lhs is the start-symbol. Otherwise it is a TDG.

T, Z, C and R are determined by the syntax of the metagrammar. I is the set of symbols which can not be defined otherwise.

G will be transformed into a process G_p by a compiler and an interpreter, as discussed in section 1.1 . The semantics of G will be determined by the (input, output) pairs of G_p.
It will depend upon the operating environment of G_p whether input and output are related to other processes, to disk-files or to strings which are provided by surrounding programs. For instance, in the Parspat system the compiler and the runsystem can be provided to a program as an external procedure written in a programming language, or they can be called from the shell of an operating system.
In general we abstract from the environment and use the name of the process G_p also as a function to denote the transformation of input into output. The sources of input and output stem from the functional environment of grammatical systems. In section 1.2 we identified a number of these sources.

We will denote the input and output of a particular process G_p by a number of parameters and write G_p(In, Lex, Rec, Pa, Ou, Rep, Bld), where
- In is the input as a string in T*,
- Lex is the lexicon which has to be used,
- Rec is a Boolean variable which will receive the value true in case the recognition succeeds, and false otherwise,
- Pa is the parse which will be created on-line; if G_p is called with ε as actual value for Pa no parse will be produced,
- Ou is the output as a string in (V+Z)* : the symbols in V* may be accompanied by variable symbols in Z with their value (this will be discussed later on); if G_p is called with ε as actual value for Ou no output will be produced,
- Rep is a string in R*; if G_p is called with ε as actual value for Rep no reports will be produced,
- Bld is a string in B*; if G_p is called with ε as actual value for Bld no builds will be produced.

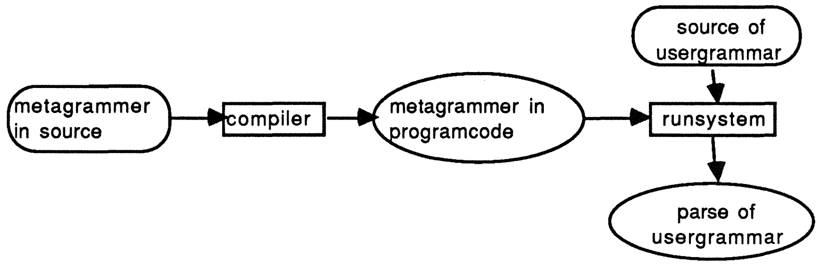
Definition.
A cascaded grammar C is a set $\{G_1, G_2,..., G_{m-1}, G_m\}$, m>1, where $G_1..G_{m-1}$ are TDG's and G_m is a TDG or a PSG.
A cascaded process Cp is a set $\{ G_{p1}(In_1, Lex_1, Rec_1, Pa_1, Ou_1, Rep_1, Bld_1),...,G_{pm}(In_m, Lex_m, Rec_m, Pa_m, Ou_m, Rep_m, Bld_m) \}$, m>1. All processes $G_{p1},...,G_{pm}$ run in parallel. The output of one process may be the input to another one. This is indicated by parameters with the same name.
Δ

In the unifying formalism six components are distinguished :
1. the BNF notation for Chomsky type-0 grammars,
2. regular expressions,
3. reserved notions for patterns, trees and references between lhs's and rhs's,
4. the use of variables,
5. notations for Boolean construct
6. notations for output.
Although the unifying formalism is a whole we prefer to discuss it according to these six components with the risk of some overlap. In the following subsections each component will be discussed by going into
a) its functional aspects as related to applications,

b) its functional aspects as related to other sub-formalisms,
c) its lexical considerations,
d) its syntax, presented as rules of the metagrammar,
e) its semantics and
f) some examples.
In trivial cases we will leave out some of these points. The semantics will be presented informally from a generative point of view. They are constructively specified in the chapters 4, 5 and 6. More elaborate examples can also be found in chapter 8.
Many grammatical formalisms (e.g. ecfg's, atn's, attribute grammars, Chomsky type-0 grammars) will be obtained as subsets of the unifying formalism. This will be exemplified in section 2.4. In section 2.5 we present the whole metagrammar.
Our discussion starts with an overview of the lexical aspects of the unifying formalism, as defined by the metagrammar.

The metagrammar

The metagrammar describes the syntax of the rules and defines the lexical symbols. It may be compiled by the compiler in order to generate a parser which analyses grammars written in the unifying formalism:



Using a metagrammar has the advantage of being able to quickly change the spelling of reserved symbols in a U-grammar. Furthermore, by supplying a more restricted (compiled) metagrammar, users can be restricted in the choice they can make of the different sub-formalisms. As such it is easy to restrict the use of a program generator to only cfg's, patternmatching, atn's, attribute-grammars, tree-transduction, and so on.

Lexical aspects.
Some of the terminals in the metagrammar are the reserved symbols for a usergrammar. By changing these terminals the user can redefine the reserved symbols of his grammar.
Reserved symbols can be divided into *reserved operands* , *operators* and *delimiters*. We will list them in the following table according to the metagrammar which we will use in this chapter. The functioning of the reserved symbols will be discussed further in the following subsections.

Name:	Symbol:	Function:
The reserved operand symbols are :		
for patterns		
DONTCARE SIGN	*	don't care
ARB SIGN	=	arb

LINE SIGN	-	line
RANGE SIGN	..	range of characters

for trees :

OPEN_TREE	(	opening tree-bracket
OPEN_LAB_TREE	:(	labeled opening tree-bracket
CLOSE_TREE	)	closing tree-bracket
OPEN_TREES	(..(	a number of OPEN_TREE's, to be matched by the number of corresponding CLOSE_TREES
OPEN_LAB_TREES	:(..(	a number of OPEN_LAB_TREE's, including their preceding labels; the number has to be matched by the number of corresponding CLOSE_TREES
CLOSE_TREES	)..)	a number of CLOSE_TREE's, to be matched by the number of corresponding OPEN_TREES

for variables :

LAST TERMINAL	%	reference to last terminal read in

The reserved operator symbols are :

NEG SIGN	`	negation
ALTERNATIVE AND	&	Boolean "and" of alternatives in a rhs
COVER SIGN	^	coversymbol, behind a nonterminal
ASSIGNMENT SIGN	:=	assign a value to a variable
EQUAL SIGN	=	within Boolean expressions : equal
UNEQUAL SIGN	\=	within Boolean expressions : not equal
CONCAT SIGN	||	within expressions of variables : concatenation

The reserved regular expression symbols are:

OPEN_REGEXPR	[	
CLOSE_REGEXPR	]	these symbols surround a regular expression
ZEROMORE	*	behind a ']' : 0 or more times the enclosed part
ONEMORE	+	behind a ']' : 1 or more times the enclosed part
ONE	1	behind a ']' : 1 times the enclosed part

The delimiters are:

REWRITE SIGN	::	rewrite a lhs to a rhs
TRANSDUCTION SIGN	<	transduction; written between lhs and rhs when |lhs| = 1
OCTAL SIGNS	< >	between the angle brackets an octal number, representing the ascii-value of a character
CONTINUATION SIGN	,	concatenation; written between parts of a rule
ALTERNATIVE SIGN	|	between alternatives
END OF RULE SIGN	.	end of a rule
COMMENT SIGN	!	comment between !'s
ACTION BRACKETS	{ }	between these brackets actions are denoted

There is one reserved nonterminal universe, which plays a role in Boolean constructs (to be discussed later).

The symbol ' is used to surround strings of characters which have to be taken literally. For instance, in 'a:c' the colon is not the delimiter COLON .
In order to improve readability spaces and returns in the metagrammar are treated as being not significant.

In the Parspat system symbols in $N \cup I$ are spelled as any combination of available Ascii characters. Terminals and intermediate symbols are single Ascii characters or lexicon symbols. Ascii-characters can be represented by their octal code between angle brackets.
Lexicon symbols are spelled as a '$'-sign followed by any combination of available Ascii characters.

In the following subsections we assume that the following symbols are declared as follows :

ALFANUM::	DEC_DIGIT I CHAR I "', <000>...<377>, "' .
ALFANUMS::	[ALFANUM]+ .
DEC_DIGIT::	0..9 .
OCT_DIGIT::	0..7 .
CHAR::	A..Z I _ I a..z .

2.2.1 The basic formalism : Chomsky type-0 grammar

a. application
The formalism of general rewriting is, among others, provided in : transformational grammars, string-replacement, graph- and tree grammars, cs-grammars for phonological rewriting and the formalisms for machine translation. Term-rewriting systems are in use in compilers and other symbol manipulation systems (e.g. Jouannaud, 1985).
In the linguistic tradition, grammars are commonly used as a means to systematically describe the sentences of a language. This can either be from a generative or an analytic point of view. In the generative case the grammar specifies the sentences by its possible derivations. In the analytic case a sentence of the language is parsed.
One of the most established formalisms for grammars is the BNF-notation. The use of nonterminals enables the sharing of common substructures and the recursion mechanism enables recursive writing.
In transduction, grammatical knowledge, expressed in the grammar, is used to perform the operations insert, delete and change on the input, resulting in the "transduced" output. The result of a single transduction remains possible input of other grammar rules, until no more rules can be applied.
One essential difference between recognition and parsing on the one hand, and transduction on the other, is that transduction lacks a start symbol (or: distinguishing symbol). The transduction varies freely over the input.

When a U-grammar G= (N, I, T, S, Z, C, R, P, M) is reduced to a Chomsky type-0 grammar the sets of symbols I, Z, C and R are empty. M will describe that the rules in P are in general of the form:

$$X_1, X_2, .. X_n :: Y_1, Y_2, .. Y_m.$$

where $X_i, Y_j \in N \cup T$.

Depending on the maximum values allowed for n and m and restrictions on the use of terminals and nonterminals four different kinds of rules may be distinguished, which give rise to the classical "types" of phrase structure grammars and the Chomsky hierarchy. Because we will refer to these four types of rules frequently we give here a definition of the different types of rules and grammars within the Chomsky hierarchy :

Type 3 ("regular rule") : n=1 and m=1, 2 .
If m=2 then Y_1 must be a terminal and Y_2 a nonterminal. X_1 is always a nonterminal.
G is called regular if all its production rules in P are regular.

Type 2 ("context-free rule") : n=1 and m>=1 .
Sometimes m=0 is allowed, i.e. the rhs may be empty. The Y-symbols may be nonterminals or terminals, X_1 always nonterminal. (The adjective "context-free" implies that X_1 may be rewritten independently of the surrounding symbols).
G is called context-free if all its production rules in P are context-free or regular; at least one of them has to be context-free.

Type 1 ("context-sensitive rule") : n <= m, n>0 .
One of the X_i's should be a nonterminal.
G is called context-sensitive if all of its production rules in P are context-sensitive or context-free or regular; at least one of them has to be context-sensitive.
G is called "truly" context-sensitive (Aho and Ullman, 1972, p.97) if each production is of the form "α A β :: $\alpha\ \gamma\ \beta$" : A may be replaced by γ only in the context α_β. In phonological applications this is written as : "A :: γ, α_β" .

Type 0 ("type-0 rule") : n>m, m>0 .
G is called type-0 if at least one production rule in P is type-0.

The first extension of the BNF formalism is that we allow the rhs's to be empty (m=0) for type-3, -2 and -0 grammars. The second extension concerns TDG's. The difference with a type-0 grammar will be explained in the sequel.

c. lexical considerations
The delimiters are already listed in 2.2 together with their function.

d. syntax
```
GRAMMAR::                  [ RULE ]+.
RULE::                     ALTERNATIVES , ['::' | < ]1, [ ALTERNATIVES ], '.' .
ALTERNATIVES::             ALTERNATIVE, [ '|' , ALTERNATIVE]*.
ALTERNATIVE::              UNIT, [ ',' , UNIT]* .
UNIT::                     NOTION .
NOTION::                   NONTERMINAL_SYMBOL | INTERMEDIATE_SYMBOL |
                            TERMINAL_SYMBOL.
NONTERMINAL_SYMBOL::           ALFANUMS.
INTERMEDIATE_SYMBOL::          ALFANUM.
TERMINAL_SYMBOL::              BASIC_SYMBOL.
BASIC_SYMBOL::             ALFANUM  |  '<' , [ OCT_DIGIT  ]+ , '>' .
```

e. semantics
Derivations and parses.

In chapter 1 we defined the semantics of the interpretation of a formalism as the (input,output) pairs which are valid for the interpreter. Four processes for interpretation were distinguished. They will now be described in more detail.

1. Generation with a phrase structure grammar G. Input is the starting symbol, output a sentence which consists of terminal symbols. The process itself consists of a stepwise derivation.

Definition.

A derivation in G is a sequence $\alpha_1, \alpha_2, \ldots, \alpha_{m+1}$, $m \geq 0$, of strings such that for each i, $1 \leq i \leq m$, there are strings $\beta_i, \gamma_i, \delta_i, \zeta_i$ such that $\alpha_i = \beta_i \gamma_i \delta_i$, $\alpha_{i+1} = \beta_i \zeta_i \delta_i$, and $\gamma_i :: \zeta_i \in P$.

Associated with each of $\alpha_1, \ldots, \alpha_m$ there must be a pair <p,r> denoting that $\gamma_i :: \zeta_i$ is the p-th rule of P and the first symbol of γ_i is the r-th symbol of α_i. Each α_i is a line of the derivation, and the process of applying a rule to one line to produce the next line is a step of the derivation.

The sequence $\alpha_1, \ldots, \alpha_{m+1}$ is said to be a derivation of α_{m+1} from α_1, and α_{m+1} is said to be derivable from α_1.

Δ

2. Recognition with a phrase structure grammar. Input is a string $w \in T^*$. Output is the answer "yes" or "no" depending on whether w can be generated by G or not. With on-line recognition this answer will be given after the reading of each symbol.

3. Parsing with a phrase structure grammar. Input and output are the same as with a recognizer, but the process will also reconstruct all possible derivations.

The following definition of a *parse* as a two-dimensional description of a set of derivations follows closely the one of Walters (1970). It generalizes the definition of a parse tree for cfg's.

Definition.

A parse of β from α is a bracketed diagram showing how β is derived from α. Such a parse is obtained from any derivation of β from α by writing down β, then bracketing the rhs of the last rule used in the derivation, writing the subject of the rule above the bracket, and associating the rule number with the bracket. The string resulting from the replacement of the bracketed symbols by those above the bracket is the penultimate line of the derivation. If this bracketing is continued until all the steps in the derivation have been utilized, the result is a parse of β as α. The set of derivations that would yield the same parse under this construction is the set described by the parse.

Δ

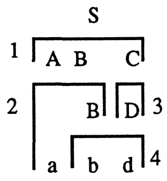
The equality of two parses can be tested after a structure-preserving transformation of a 2-dimensional bracketed parse into a string-representation.

Example. In the phrase structure grammar

(1) S :: A, B, C.
(2) A, B :: a, B.
(3) C :: D.
(4) B, D :: b, d.

a derivation and the corresponding parse are

S	<1,1>
A, B, C	<3,3>
A, B, D	<2,1>
a, B, D	<4,2>
a, b, d	



Definition.
Corresponding to every parse is a unique leftmost (rightmost) derivation, which can be constructed from the parse as follows.
Write down α. Find the leftmost (rightmost) bracket that has no brackets above it, and delete it. The next line of the derivation is determined by <p,r>, where p is the rule number labeling the deleted bracket and r is the position of the leftmost (rightmost) bracketed symbol. Repeat this process until all brackets have been deleted.
Δ
The derivation in the example above is the rightmost one. The leftmost derivation is

S	<1,1>
A, B, C	<2,1>
a, B, C	<3,3>
a, B, D	<4,1>
a, b, d	

The Parspat system delivers a parse in the form of a string from which the 2-dimensional bracketed parse can be reconstructed. For our example this string is :
$S\ {}_1(\ A\ {}_2[\ a\ B\ {}_4[\ b\ d\ ]_4\ ]_2\ B\ C\ {}_3(\ D\ )_3\ )_1$
The brackets are labeled with the number of the rule. The type-1 and -0 rewritings are indicated by square brackets. The rhs of a rule which is used in a derivation is attached to the first symbol of its lhs. This accounts for the 1-dimensionality. With the aid of the number of the rule it can be reconstructed for the 2-dimensional parse how far its brackets have to extend to the right (when the size of the lhs is fixed).
This representation may serve as a canonical representation for a 2-dimensional parse. It works for all types of Chomsky-grammars. With it the equality of parses may be tested.

4. <u>Transduction with a transduction grammar G</u>. Input is a string $\alpha_1 \in T^*$, output a string $\alpha_{m+1} \in (T+I)^*$. α_{m+1} is said to be a translation of α_1. In a cascaded grammar $C=\{G_1, G_2,..., G_m\}$ the strings α_i are element of $(T+I)^*$ for every G_i ($2 \le i \le m$).
The definition of a transduction is nearly the same as that of a derivation. The only difference is that the lhs and the rhs of the rules are interchanged.
Definition.
A transduction in G is a sequence $\alpha_1, \alpha_2,..., \alpha_{m+1}$, m>=0, of strings such that for each i, $1 \le i \le m$, there are strings $\beta_i, \gamma_i, \delta_i, \zeta_i$ such that
$\alpha_i = \beta_i \gamma_i \delta_i$, $\alpha_{i+1} = \beta_i \zeta_i \delta_i$, and $\zeta_i :: \gamma_i \in P$.
Δ
Definition.
α_m is a normal form of α_1 if α_m is a translation of α_1 and no rule is applicable to α_m.
Δ
We repeat from Dershowitz (1985) three desirable properties for transduction grammars :
(1) termination- no infinite derivations are possible
(2) confluence- each string has at most one normal form

(this is also called the Church-Rosser property)

(3) soundness- equal strings are only rewritten to equal strings, that is, there is only one possible derivation; we call this property also "unambiguous".

Each of these properties is in general undecidable. Dershowitz (1985) shows that confluence is decidable for terminating systems.

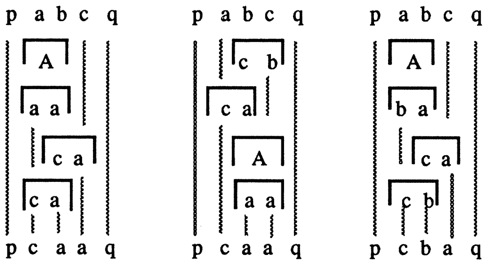
We are interested in the automatic construction of transducers for general transduction grammars, whether these grammars have the desirable properties or not, and in a classification of heuristics, stemming from applications, which may be built into the transducers in order to obtain one or more of the desirable properties.

The following example shows some transductions with the ambiguous (and therefore not sound) transduction grammar:

(1) a, a :: A.
(2) b, a :: A.
(3) c, a :: a, c.
(4) c, b :: b, c.
(5) A :: a, b.

and the input string α_1 = pabcq.

The transduction process according to this grammar terminates, is not confluent and is not sound. (Transductions 1 and 2 do not reject confluence because they rewrite to the same normal form, but 2 and 3 do; 1, 2 and 3 reject soundness because equal terms are rewritten to different terms.)



(1) (2) (3)

Central in the processes for recognition and transduction is the (re)construction of a derivation. In this book we study the on-line aspects of these processes and restrict the interpretation to on-line interpretation.

An on-line interpretation consists of an on-line (re)construction of a derivation and of the creation of the normal form(s) within a finite delay after the reading of the input string, where input and output possibly overlap in time with each other.

f. example

a csg which generates the language $\{a^n b^n c^n d^n \mid n \geq 1\}$ (Levelt,1973):

Z :: E , Z , F | a , b , c , d .
E , a :: a , E .
d , F :: F , d .
E , b :: a , b , b .
c , F :: c , c , d .

2.2.2 Regular expressions

a. application
If regular expressions are used in the rules of a cfg then the grammar is called an ecfg. Ecfg's are quite naturally used in phonological transduction, morphological analysis and pattern matching. Regular expressions provide for a compact notation of repetition and embedded alternation. Syntax-diagrams and recursive transition networks (see also section 2.3.1.4) are easily rewritten into regular expressions. Regular expressions are defined in e.g. (Hopcroft and Ullman, 1979).

b. functioning within formalism
Regular expressions could be used in the lhs.

c. lexical considerations
The reserved regular expression symbols were listed in section 2.2

d. syntax
```
UNIT::              ENCLOSEDPART .
ENCLOSEDPART::  '[' , ALTERNATIVES , ']', [ '1' |  '*' | '+'].
```

e. semantics
A notion is the most simple regular expression. If r, r1 and r2 are regular expressions then the following ones are also regular expressions :

expression		meaning
r1 , r2	:	concatenation of r1 and r2
r1 | r2	:	r1 or r2
[r]*	:	0 or more repetitions of r
[r]+	:	1 or more repetitions of r
[r]	:	0 or 1 r.
[r]1	:	1 r.

A regular expression can be rewritten as a weak equivalent cfg, which means that the same language will be generated but not with the same derivations. In the parse of a regular expression no special marker will signal the appearance of the expression.

f. example
```
        [ a | b]1 , [ c , d | [ E ]+ ]* , [ f ] .
```
is a regular expression.

2.2.3 Notions for pattern matching

2.2.3.1 Don't care

a. application
In a number of applications the notions "don't care" and "arb" are necessary. In pattern matching they are used as "wild-card's", in syntactic analysis as "any intervening material". In chapter 1 we indicated that "don't cares" and "arb's" may also appear in texts.

d. syntax
The same as for a notion in a rhs.
```
TERMINAL_SYMBOL::     '*' .
```

e. semantics
A "don't care" generates an arbitrary terminal or an intermediate symbol.

f. example
Code :: '<', *, *, *, '>'.
Code is defined here as as a sequence of three arbitrary characters, surrounded by angle brackets.

2.2.3.2 Arb

a. application
The "arb" is named after the built-in function "arb" of Snobol. It matches a number of characters up to the character(s) which may be expected after the arb, according to the grammar.

b. functioning within formalism
The functioning of the "arb" ends with the closing bracket of the current tree-level (which will be further explained in the section on tree symbol).

d. syntax
The same as for a notion in a rhs.
TERMINAL_SYMBOL:: '=' .

e. semantics
The "arb" generates any sequence of terminal and intermediate symbols, possibly empty, but excluding the symbols which may follow the "arb" in the derivation.

f. examples
1. Comment :: '!', =, '!'.
A sequence of characters which starts with an exclamation mark and which ends with an exclamation mark is recognized as Comment. The characters which are matched by the "arb" will not contain a '!'.
2. Sentence :: =, '.' .
Sentence is defined as the maximal character sequence not containing a dot, followed by a dot. Note that with such a definition an input such as: 'Dr. West reads his mail.' will be matched by the nonterminal Sentence upto the dot in 'Dr.' .

2.2.3.3 Line

a. application
The "line" plays an essential role in the unification of parsing and pattern matching. It allows for the multiple matching of phrases in an input.

b. functioning within formalism
The functioning of the "line" ends with the closing bracket of the current tree-level (which will be further explained in the section on tree symbols).

d. syntax
The same as for a notion in a rhs.
TERMINAL_SYMBOL:: '-' .

e. semantics
The "line" generates any sequence of terminal and intermediate symbols, possibly empty, including the symbols which may follow the "line" in the derivation.

f. example
  Ing_form :: -,'ing'.
Ing_form is defined as a sequence of characters, ending in 'ing' and possibly including 'ing'. The "line" and the "arb" are not exchangeable in this case : if the input is 'singing' then Ing_form will match twice as 'sing' and 'singing'. If instead of the line an "arb" had been specified then only 'sing' could match.

2.2.3.4 Range of terminal symbols

a. application
On the lexical level ranges of terminal symbols may be useful, for instance in morphological and phonological grammars and grammars for the description of texts.

c. lexical considerations
The range consists of 2 dots between two terminal symbols.

d. syntax

NOTION::	USER_NOTION .
USER_NOTION::	RANGE_SYMBOL, '..', RANGE_SYMBOL .

e. semantics
In the Parspat system the range functions as an abbreviation for an ordered subset of the ASCII character set. The range includes the two terminals immediately surrounding it plus those in between them, defined by the natural order of the character set.

f. examples
  Consonant :: b..d | f..h | j..n | p..t | v..x | z.
  Ascii :: <0>..<377> .

2.2.4 Notation of Actions

a. application
The extensions to the formalism for variables, for Boolean constructs between rules and for output are all denoted within braces. These extensions will be treated in the following subsections. In this subsection we will discuss where the actions may be placed and when they are executed.

b. functioning within formalism
The actions may be placed before a ',', a '|', a ']' and a '.' .

c. lexical considerations
Between the braces '{' and '}' actions are denoted.

d. syntax
We extend the syntax of regular expressions in section 2.2.2 in order to formulate the admissible places of actions within regular expressions.
ENCLOSEDPART:: '[' , ALTERNATIVES , ']',

```
                [ ACTIONS], [ DEC_DIGIT | '*' | '+'], [  ACTIONS ] .
ALTERNATIVE::   UNIT, [ACTIONS], [ ',', UNIT, [ACTIONS] ]* .
ACTIONS::       '{', ACTION ,  ['&' ,  ACTION ]*, '}'.
ACTION::        REPORT_DECLARATION | COOP_DECLARATION |
                assignment | TEST | BUILD_DECLARATION |
                GRAMMAR_CALL | PROCEDURE_CALL .
```

e. semantics
The syntax of actions is embedded within the syntax of regular expressions. The semantics concern the contribution of actions to input and output. This will be dealt with in the following subsections. It is here the place to discuss at which moment during a derivation actions are executed.
During a derivation actions are executed between the writing of 2 symbols. In the following table we define which actions will be executed for all possible situations. The dot represents the position in the regular expression during the rewriting.

$\alpha\,.a\ \{A1\}\ \beta$	-a->	$\alpha\ a\ \{A1\}\ .\beta$: execute A1
$\delta\ [...\mid \alpha\ .a\ \{A1\} \mid ...]\{A2\}\ \beta$	-a->	$\delta\ [...\mid \alpha\ a\ \{A1\} \mid ...]\ \{A2\}.\beta$: execute A1 and A2
$\delta\ [...\mid \alpha\ .a\ \{A1\} \mid ...]\{A2\}^*\{A3\}\ \beta$	-a->	$\delta\ [...\mid \alpha\ a\ \{A1\} \mid ...]\{A2\}^*\ \{A3\}.\beta$: execute A1, A2 and A3 (idem for]+)
$\delta\ [\ \gamma \mid \alpha\ .a\ \{A1\} \mid ...]\{A2\}^*\{A3\}\ \beta$	-a->	$\delta\ [\ .\gamma \mid .\alpha\ a\ \{A1\} \mid ...]\{A2\}^*\ \{A3\}\beta$: execute A1 and A2 (idem for]+)
$\delta\ .a\ \{A1\}[\ \alpha\ ]\{A2\}^*\{A3\}\ \beta$	-a->	$\delta\ a\ \{A1\}[.\ \alpha\ ]\{A2\}^*\ \{A3\}\beta$: execute A1 (idem for + and 1 and for the absence of *, + and 1)
$\delta\ .a\ \{A1\}[\ \alpha\ ]\{A2\}^*\{A3\}\ \beta$	-a->	$\delta\ a\ \{A1\}[\ \alpha\ ]\{A2\}^*\ \{A3\}.\beta$: execute A1 and A3 (idem for the absence of *, + and 1)

f. example
In order to illustrate the activation of actions in combination with regular expressions we give the following grammar as an example. The actions between braces are abbreviated by A1, A2, A3 and A4.

```
        S :: T {A4} .
        T :: a , [T {A1}]* {A2} , a {A3} .
```

The input 'aaaa!' will give rise to the following parse tree and sequence of actions :

```
        S :( T :( a T :( a {A2} a {A3} ) {A1,A2} a {A3} ) {A4} ).
```

2.2.5 Variables

a. application
Variables are present in a number of grammar-formalisms. They may be divided into three classes :
- formal parameters bound to nonterminal symbols (like attribute grammars and Prolog),
- global variables, like those used in atn-grammars (scope: the whole grammar),
- local variables, (scope: the grammar rule in which they appear).

The possible operations on variables may be summarized by :
- Snobol-like conditional-assignment to variables from the text,
- assignment of expressions of other variables,

- tests on (expressions of) variables; a desirable feature (for unification) is postponed evaluation and assignment when variables do not have a value,
- matching on the value of variables and assigning substructures to other variables
- interaction with user-defined external routines.

The only feature which is not allowed in the unifying formalism is the use of global variables. The consequence for ATN grammars is that the variables which are used in a sub-ATN have to be passed by parameters.

b. <u>functioning within formalism</u>
Nonterminals and lexicon terminals may be enriched with parameters. The parameters of nonterminals at a lhs have to be preceded by the declarations "I:", "O:" or "IO:", which correspond respectively with the features "inherited", "synthesized" and the combination of both, which are familiar in attribute- and affix-grammars. Parameters of lexicon terminals need the "O:" declaration. In the latter case the actual value(s) will be supplied by the lexicon.
The value of a variable is a string of arbitrary length. Within an expression variables may be concatenated, together with string-constants and the last read-in character.
A variable is declared by its first appearance in a rhs. The scope of the variable is the grammar rule with that rhs.
Operations on and with variables are denoted within the action-brackets. The operations are : assignment, test, grammar-call and procedure-call.
In an assignment, an expression with variables and constants is assigned to a variable.
A test concerns the truth-value of a Boolean expression. If this value is "false" then the current recognition path stops.
A grammar-call consists of the name of a grammar with the seven parameters (In, Lex, Rec, Pa, Ou, Rep, Bld) which we discussed in section 2.2.
In the output a number of variables with their values may appear which will be bound to variables with the same name in the current grammar rule. The name of the grammar can be external.
A procedure-call consists of the name of an external procedure which is written in some programming language, and an unlimited number of parameters in which (expressions of) variables may be passed. The first parameter is a Boolean. If it returns the value "false" then the current recognition path stops.

c. <u>lexical considerations</u>
Variables are denoted as strings of characters and digits. The concatenation operator is '||'. A string-constant is a string of Ascii-characters between quotes. In the string the symbols '(' and ')' are reserved and act as tree symbol. The reserved symbol '%' stands for the last symbol that is read in.

d. <u>syntax</u>

```
NONTERMINAL_SYMBOL::ALFANUMS, PARAMETERLIST.
PARAMETERLIST::         '(',  F_OR_A_PARAMETERS  , ')'  |
                        '('  ,  ACTUAL_PARAMETERS  , ')'  .
F_OR_A_PARAMETERS::  PARAMETER_DECL , [',', PARAMETER_DECL]*  .
ACTUAL_PARAMETERS:: ACTUAL_NAME  ,  [',',  ACTUAL_NAME  ]*.
PARAMETER_DECL::        [ I | O | I , O ]1, ':' , ALFANUMS  .
ACTUAL_NAME::           ALFANUMS .
ASSIGNMENT::            ALFANUMS ,   ':=' , VARIABLE_EXPR  .
TEST::                  VARIABLE_EXPR  , [ '=' |  '/= ']1 , VARIABLE_EXPR  .
VARIABLE_EXPR::         VARLIT , [ '||' ,  VARLIT]*.
VARLIT::                ALFANUMS | LITERAL  | '%'  .
```

```
GRAMMAR_CALL::          ALFANUMS, '(', [ ALFANUMS, ',']6, ALFANUMS , ')' .
PROCEDURE_CALL::        ALFANUMS, PARAMETERLIST.
LITERAL::                "' , =  , "' .
```

d1. restriction in Parspat

Unification of variables will be implemented in a later stage. In the current implementation variables which appear in an expression need to have a value assigned to them when the expression is evaluated.

e. semantics

The semantics of variables are determined by the way in which operations on variables influence the output stream and by their inhibition of a current parse. These operations are straightforward and work in the same way as in programming languages like Pascal.

f. examples

See also chapter 8 for more elaborate examples.

1. assignment to a variable from the text :

```
S(O:A,O:B) :: [a..g {A := A || %} | h..z {B := B || %} ]*.
```

After recognition of the input all characters in the range a..g are appended in variable A and all characters in the range h..z are appended in variable B.

2. assignment and tests with variables and constants :

```
S(NGETAL)::          NP(NGETAL), VP(VGETAL) {NGETAL=VGETAL}.
NP(O:Ngetal)::       [LIDW], [BIJVNW]*, NAAMW(Ngetal).
VP(O:Vgetal)::       FINWW(Vgetal), [INFINWW]*.
LIDW::               'de' | 'het' | 'een'.
BIJVNW::             'groen',[e].
NAAMW(O:N)::         'mannetje' {N:='E'}, [s {N:='M'}].
FINWW(O:N)::         'wa', [s {N:='E'} | 'ren'  {N:='M'}]1.
INFINWW::            'gevonden'.
```

After recognition the name of the variable NGETAL with its value will be brought to the output.

3. assignment from a lexicon :

NP:: $det(gender), $adj(singplu1), $noun(singplu2) {singplu1 = singplu2 }. We assume that the lexicon contains the value for the gender of a "det" and the multiplicity of an "adj" and of a "noun".

4. Call of an external procedure :

```
S ::    -, NP(singplu1, time1, head) { Sem(O:continue, I:time1, I:head, IO:expect) },
           VP(singplu2, time2, expect) { singplu1 = singplu2; time1 = time2 }.
```

After the recognition of an NP the routine "Sem" is called with as input parameters "time1" and "head" and as output parameters "continue" and "expect". If the value of "continue" upon return of Sem is "false" then further recognition of this path is inhibited (there may be more paths active because this is a pattern grammar). Else recognition continues with the nonterminal VP which gets the variable "expect".

2.2.6 Booleans

2.2.6.1 Boolean "and" between rules

a. application

Especially in the case of pattern grammars it is desirable to intersect grammar rules. In the application of Corpus Linguistics, which we discussed in chapter 1 and for which a number of examples are presented in chapter 8, sentences and parse trees in corpora are queried with Boolean combinations of patterns. These patterns have, in general, the form of a rhs in the unifying formalism. In order to accommodate for Boolean combinations of patterns we introduce the concept of the intersection of rules.

Suppose we want to match the patterns P1 and P2, which are written in the formalism of a rhs, with a sentence. For the matching of the Boolean "or" of P1 and P2 we write in the unifying formalism grammar G1: S :: -, P1, - | -, P2, - . For the Boolean "and" we introduce the reserved symbol "&" and write the grammar G2: S :: -, P1, - & -, P2, -. (The Boolean "not" will be introduced in the following subsection.)

The treatment of the intersection of rhs's can be generalized further along the following lines. Grammar G2 differs from grammar G1 in that respect that for G2 a check has to be made whether both P1 and P2 match. We introduce an alternative notation with which we express that requirement in G1, which then becomes grammar G3 : S :: -, P1, - {C:abc} | -, P2, - {C:abc}, with "abc" as a cooperation symbol (which name may be arbitrary chosen), written between the braces of an action. Semantically the two grammars G2 and G3 are equivalent. But we allow also that the cooperation symbols may be written at arbitrary places in a rhs, as an action. For instance, consider the grammar G4 : S :: -, P1, - {C:A}, P3, - {C:B} | -, P2, - {C:A}, P4, - {C:B}. Here we express that S generates strings w = u v such that u is generated by -, P1, - as well as by -, P2, - and that v is generated by -, P3,- as well as by -, P4,- . From the recognition point of view we may say that the recognition of P3 and P4 has to wait for the recognition of P1 and P2, and that at least at the end of the input both alternatives of the rhs have to be recognized.

This resembles the cooperation of parallel processes. In G4 the 2 rhs's may be waiting for each other at the rendez-vous A, consuming input (which is possible because of the appearance of lines, which may recognize an indefinite number of input symbols).

In general, when all rhs's which contain the same cooperation name are recognized up to the location where the name is denoted as a cooperation symbol then all these rhs's may continue, otherwise they all fail.

The device of cooperating rules is especially useful in the specification of complex conditions in trees. Several tree-walking languages do exist in which it is possible to state in a dynamic way how to arrive from one node in a tree to another node, and to specify conditions on the labels at these nodes. With the device of cooperation symbols it is possible to translate dynamic tree-walking rules into static ones.

If instructions for tree-walking take the following form:

> "If at a position A condition C1 obtains, then move through the tree following path P to position B. There, condition C2 should obtain".

then these instructions can be stated in a static form as (De Jong and Masereeuw, 1987):

> "If in the tree there is a position A where condition C1 obtains, then there should also be a position B where condition C2 obtains, and moreover there should be a positional relation between A and B that can be expressed as P".

The specification of a path P between A and B can be the same as for a path P' between positions A' and B'. If this happens frequently it may be appropriate to abbreviate this, in a tree-walking language, by a procedure. The corresponding method in the unifying formalism is to use a nonterminal as an abbreviation, supplied with two cooperation points as parameters. We refer further to section 2.3.1.1 for an appropriate example.

b. functioning within formalism

The name of a cooperation appears within the braces as an action. The place among other actions written between the same braces is not relevant. An indefinite number of cooperation symbols may be written between the same braces. The name of a cooperation has to appear in two or more rhs's in order to become meaningful. These rhs's do not necessarily need to have the same lhs : a lhs may be a nonterminal which is used for abbreviation purposes.

c. lexical considerations
A cooperation symbol is written in the action-part and consists of a string of alphanumeric characters. When a cooperation appears only at the end of alternatives within one and the same regular expression then it can be replaced by an '&' instead of the alternative sign '|' between the alternatives.

d. syntax

ALTERNATIVES::	ALTERNATIVE, ['&' , ALTERNATIVE]*.
PARAMETER_DECL::	COOP_DECLARATION .
COOP_DECLARATION::	C, ':', ALFANUMS .

d1. restrictions in Parspat
The use of the intersection of grammar rules originated from the application of fast pattern matching in large text corpora . Up till now we implemented in the Parspat system the cooperations in combination with those sub-formalisms for which a FSA can be constructed. This will be dealt with further in the chapters 5 and 6.

e. semantics
The informal semantics have already been expressed in a. The formal semantics are simply expressed with the aid of the construction of itemsets, which happens in the chapters 5 and 6.

f. examples (from De Jong and Masereeuw, 1987)

```
        S :: -, Subject, -      {C: SVO}.
        S :: -, Verb, -         {C: SVO}.
        S :: -, Object, -       {C: SVO}.
```

This is equivalent to:

```
        S :: -, Subject, - & -, Verb, - & -, Object, -.
```

Equivalent grammar without cooperation:

```
        S :: -, Subject, -, Verb, -, Object, -.
        S :: -, Subject, -, Object, -, Verb, -.
        (etcetera for all 6 permutations).
```

An elaborate example of the use of cooperation points as parameters will be given in section 2.3.1.1.

2.2.6.2 Boolean negation within a rule

a. application
The Boolean negation (complementation) is desirable with "rewriting in context" (phonological rewriting) and in query-languages for free-text database systems. In (Aho, Hopcroft and Ullman, 1974, p. 419) it is argued that Boolean operators in regular expressions may shorten their length, and therefore may provide for a more compact notation.

b. functioning within formalism
Boolean negation functions throughout the unifying formalism. It is therefore appropriate to formulate its effect on the other sub-formalisms. This will happen in subsection e.

c. lexical considerations

reserved symbols :	1. the backquote `
	2. the nonterminal UNIVERSE.

d. syntax : ! on regular expressions: backquote before the open bracket !

```
ENCLOSEDPART::  '`[' , ALTERNATIVES , ']',
                 [ ACTIONS], [ '1' | '*' | '+'], [ ACTIONS ] .
                ! on notions: backquote behind the notion !
NOTION::        [ NONTERMINAL_SYMBOL | INTERMEDIATE_SYMBOL |
                 TERMINAL_SYMBOL ]1, '`'.
```

d1. restrictions in Parspat

1. Up till now we have implemented in the Parspat system the negation operator in combination with those sub-formalisms for which a FSA can be constructed. This will be dealt with further in chapter 5.

2 We are not aware of the applicability of the negation-operator in a lhs and therefore have not implemented it for a lhs.

e. semantics

Boolean negation and symbols

- terminal and intermediate symbols

The negation of a terminal is defined as the complement of the set of which it is a member. This set can be defined with the aid of the reserved symbol "UNIVERSE".
A universe can be defined by inclusion of the following rule in the grammar:

UNIVERSE:: <definition of universe> .

In the Parspat system, if no universe is defined, the rule "UNIVERSE:: <000>..<377>" will automatically be added. If for the compiler of the Parspat system the switch SHARED_UNIVERSE is set to true then the set of intermediate terminals I will be added to the universe. When this switch is false then there will be separate universes for input terminals and intermediate symbols. Negation in this case, won't cover intermediate symbols. With this definition the negation of a "don't care", an "arb" and a "line" does not exist.

- strings of terminal and intermediate symbols

The negation of a string $x \in (T+I)^*$ is defined as the set of strings $\{w \mid w \in (T+I)^*, |w| = |x|, w \neq x\}$. That is, x` generates all strings w with the length of x, but not equal to x.

- nonterminal symbols

The negation of a nonterminal A is defined as the set of strings $\{w \mid w \in (T+I)^*, |w| = |x|, w \neq x, A =*> x\}$. That is, A` generates all strings w with length between the length of the shortest and the largest string that can be generated by A, but not equal to any string that can be generated by A.

Boolean negation and regular expressions

We define the interpretation of `[α}, where } means ],]1, ]* or ]+, as the same as of `[[α}]1. If we substitute the nonterminal A for [α} then this expression becomes A`, which we discussed above.

f. examples

a`	an arbitrary character, but not an 'a'
`[a,b,c]1	a sequence of 3 characters, which is not 'abc'
A`	where A is a nonterminal. A` matches any sequence of the same length as one of the sentences generated by A, which is not equal to such a sentence

`['ab'|'bc']1	a sequence of 2 character, which is not 'ab' and not 'bc'
`[-,a..c]1	any sequence of characters which does not end in 'a', 'b' or 'c'
`[-,[A|B]1,C-]1	any sequence of characters which does not contain a sentence which may be generated by AC or BC
NP:: -, [Adj`], N, -	a Noun Phrase with a Noun, but without an Adjective preceding that Noun.

The following rules are part of a phonological grammar which is shown in chapter 8.

```
        c, K, NOTeORiORh            :: c, NOTeORiORh.
        NOTeORiORh                  :: `[ e | i | h ].
        UNIVERSE                    :: a..z.
```

The 'K' in the first rule is an intermediate symbol (it appears without being present in the rhs). It is essential that SHARED_UNIVERSE = FALSE, because otherwise 'K' would fall under the definition of NOTeORiORh, and the first rule would be applicable to its own result indefinite.

2.2.7 Input

2.2.7.1 Tree symbols

a. application

In a number of applications we met the datastructure of a tree: in the structure of texts, in parse trees, in the formalisms for machine-translation. It is such an evident datastructure that it has to be supported by the unifying formalism.

b. functioning within formalism

We distinguish labeled and unlabeled trees, both in patterns and in texts. The notion before the open parenthesis of a labeled tree functions as a label. A labeled tree in the input may be skipped when the tree is not specified in the grammar. An unlabeled tree may not be skipped.

c. lexical considerations

The distinction between labeled and unlabeled trees is made by a colon before the open parenthesis. To descend to an arbitrary level in a tree, one can use the reserved nonterminal '(..('. To climb up to the level of the corresponding '(..(' the reserved nonterminal ')..)' can be used.

d. syntax

```
UNIT::              TREEUNIT | TREESUNIT.
TREEUNIT::          [ '(' | NOTION, ':(']1,   ALTERNATIVES, ')', [ ACTIONS ].
TREESUNIT::         [ '(..(' | NOTION, ':(..(']1,   ALTERNATIVES, ')..)',
                     [ ACTIONS ].
```

e. semantics

The opening tree-bracket and the labeled opening tree-bracket do not match with each other. The label before a labeled opening tree-bracket has the form of a notion, with all possibilities for the expression of e.g. a Boolean negation or a range. In essence the two opening brackets differ in the following way :

- a labeled opening tree-bracket requires a notion
- a labeled opening tree-bracket in the input will be skipped up to its corresponding closing bracket when the opening tree-bracket in the input is not expected.

The process of skipping can be speeded up by placing a pointer in the text at a ':(' which points to the corresponding ')'. This happens when files are created with the aid of the "build" operator.

f. examples

1. If the start of the input text is :

```
 BOEDEL :( KLASSE :( D )
       GEZIN :( G )
       ORG RN :( HUIS KAT :( VR :( VO :( HUIS )
                                   PR :( WOON )
                                 )
                          )
                  LAND KAT :( VR :( VO :( TUIN )
                                 )
                          )
                  EFFECT KAT :( VR :( VO :( OBLIGATIE )
                                    BIJZ :( FRANKRIJK )
                                    AA :( 1 GELD )
                                    HOEV :( GT :( 2 )
                                           )
                                )
                              TAX :( 2 )
                              VR :( VO :( OBLIGATIE )
                                    BIJZ :( PLANTERS OP DE EILANDEN )
                                    RENTE :( 4 )
                                   )
                              TAX :( 1 )
                             )
            )
...
```

and the grammar is

```
S :: - , BOEDEL :(- , ORG , RN :( [A & B & C]1 ), - ) .
A :: - , HUIS , KAT , - .
B :: - , LAND , KAT , - .
C :: - , EFFECT , KAT :(- , TAX :( * ) , - ) , - .
```

then the input will be recognized, and '1' will be matched by the '*'.

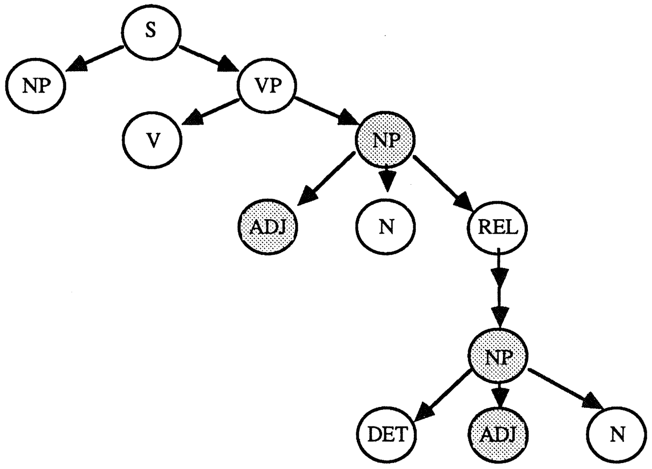
2. The second example concerns the matching at arbitrary levels in a parse tree.

If the tree is :

```
S:( NP,
   VP:( V,
       NP:(ADJ,
          N,
           REL:(...
                 NP:(DET,
                    ADJ,
                    N)
              ...)
          )
       )
   )
```

and the pattern grammar is :
ADJ-NP :: S: (..(, -, NP:(-, ADJ, -,), -,)..).

then the result will be two matches in the parse tree as shown below :

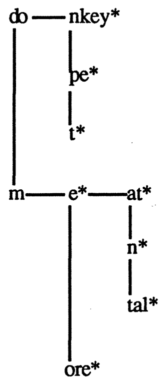


2.2.7.2 The lexicon

<u>a. application</u>
As was already stated, terminals may be single ASCII characters or lexicon symbols, which are preceded by a '$'. The Parspat runsystem cooperates with a lexicon with the datastructure of a trie.

We will first give an example of a trie (supplied by J. Skolnik). A star denotes the end of an entry.

Trie :



Contents of the trie :

donkey
dope
dot
me
meat
men
metal
more

There are a number of properties of tries which make them useful for the automatic processing of text. They are :

- the alphabetical order of entries is preserved
- common prefixes of entries are stored only once
- the lookup of an entry is an on-line process : during the reading of an entry from the input one may walk in parallel through the trie, getting an indication of whether further progress is possible
- because of this on-line property it is a good companion for on-line syntactic analysis
- multi-words may be stored and processed in a simple way
- entries may have any length; no fixed storage has to be reserved.

Kunst and Blank (1982) discuss the merits of the trie-datastructure for morphological analysis.
In his survey on access methods for text, Faloutsos (1985) categorizes the trie as the only datastructure which enables access to a list of keywords in a time which is proportional to the length of the keyword searched for. However, if the trie has to be made external on disc, a more than trivial implementation is necessary in order to let it compete with the widely known family of B-trees and with hash-coding methods. Because of the absence of such a non-trivial implementation Faloutsos further ignores tries on external memory.
The Parspat runsystem makes use of an efficient implementation of such an external trie (Skolnik, 1982). In particular the number of disc-accesses which are needed for the lookup of an entry is optimized. Each position in the trie may be characterized by a set of internal pointers. Functions exist to ask for the current position, to store a position and to locate on a position. By storing positions in the trie itself the external trie may be transformed into an external dag or an external network.

The "insert" and "delete" operations on a lexicon may be performed as an action. The "lookup" operation is performed on-line and will be initiated when a lexicon terminal in the grammar has to be matched. After the reading of the next character three messages may be reported by the lexicon function: failure (no such entry), success (entry found) and proceed (continuation possible). The last two messages may be combined when an entry is found which is the prefix of another entry. In the case of failure the current derivation will stop.
Entries in the lexicon are strings of arbitrary length and may be followed by one or more lexical notions, separated by a delimiter. The categories will be successively assigned to the

variables which are denoted as parameters with the lexicon terminal. The first parameter will always return the matched entry.

b. functioning within formalism
The incremental lookup operation will operate on the lexicon which was named in the call to the grammar. The "insert" and "delete" operations may specify, optionally, the name of another lexicon.

c. lexical considerations
Entries in the lexicon may contain any printable character except '#', which acts as a delimiter. The lexical notions should have exactly the same spelling as in the grammar, including the '$'. Lexical notions that do not occur in the grammar are ignored.

d. syntax
concerning the construction of the lexicon :
```
LEXICON_STRING::      ENTRY, [ '#', CATEGORY]* .
ENTRY::               [ ALFANUM | SPACE ]+ .
CATEGORY::            [ '$', ALFANUM]+ .
```
concerning the grammar :
```
TERMINAL_SYMBOL::     "$", ALFANUMS.
action::              'update(', LEXICON_STRING, [ LEXICON_NAME ], ')' |
                        'delete(', LEXICON_STRING, [ LEXICON_NAME ], ')' .
LEXICON_STRING::      VARIABLE_EXPR, [ '#', VARIABLE_EXPR]* .
```

d1. restrictions in Parspat
Entries are case-insensitive, i.e. they are converted to lowercase.

e. semantics
When during generation a lexicon symbol is encountered (eventually with parameters) then one of the ENTRY's in the lexicon is selected with as its first CATEGORY the lexicon symbol. Its other CATEGORY's have to correspond with the parameters.
The semantics for on-line recognition according to a grammar can be extended to the on-line processing of a lexicon which has the trie-datastructure. By combining these semantics the external lexicon can be treated as an integral part of the grammar.

f. example
After execution of the actions
```
        {...  update('work#' || v1) ...}  and  {...  update('work#' || v2) ...}
```
with variables v1 = '$V' and v2 = '$N'
and a lexicon which contains only
```
        working*#$Adj
```
the lexicon will contain :
```
work*ing*#$Adj
    |
    #$N
      |
      V
```
(because of the trie-structure).
If during parsing of a rhs the notion '$V' is encountered and the input contains 'work' then the derivation will be continued.

2.2.8 Output

2.2.8.1 Reports

a. application
"Reports" are used in order to trigger other processes during recognition. They enable a "syntax-driven" approach with a strict separation between syntax and semantics. The following remarks concern on-line recognition. The reports will be brought to the output string as soon as possible. If there is only one derivation this means : immediately. When there exist two or more derivations the reports are stored within the parse. At the moment that some derivation stops and only one derivation exists then the stored reports are brought to the output string: in that respect the outside environment is not aware of temporary ambiguities, but responses may be delayed.

b. functioning within formalism
There are two report functions. The "S"-function (for "signal") brings to the output-string the number which follows the S. The "R"-function (from "report") does the same, but appends to it the last terminal or intermediate nonterminal symbol(s) which is (are) read in. There may be more symbols in case the notion before the action where the report-function is denoted is an arb ('=') or a line ('-'). In that case all the symbols which are covered by that notion are concatenated and reported (note that this string may also be empty). Consecutive reports with the same number are merged into one report with all symbols concatenated.

d. syntax
```
REPORT_DECLARATION::        [R | S]1 , ':' , [ DEC_DIGIT  ] + .
```

e. examples
1. pattern matching with 4 keywords, signaling a match at the end of a keyword.
```
        A :: -,'he' {S:1},- | -,'she' {S:2},- | -,'his' {S:3},- | -,'hers' {S:4},- .
```
The same result is obtained by writing :
```
        A :: -,'he' {S:1},'rs' {S:4},- | -,'she' {S:2},- | -,'his' {S:3},- .
```
If the input is 'ushers' then the output report string will be '1<NL>2<NL>4<NL>', where <NL> is the symbol for a carriage return/new line.
```
2.      B :: C {R:1}, = {R:2}, [d, * {R:3} ]+, e .
        C :: c | f .
```
If the input is 'cabcdededee' then the output report string will be '1c<NL>2abc<NL>3eee<NL>'.   !

2.2.8.2 Structural description

a. application
We already discussed the representation of a parse as a structural description. In the case of parsing an ambiguous grammar all parses have to be brought to the output parse string.

b. functioning within formalism
The parse(s) is (are) created during recognition. Each nonterminal will cover a sub-parse tree. This sub-parse tree can be assigned to a variable within an actionpart by simply writing the nonterminal. On the other hand a variable, containing a (possibly modified) (sub-)parse tree, can also be assigned to a nonterminal.

c. lexical considerations
Parses are created when in the call to the grammar the actual value for the parse parameter is not the empty string.

d1. implementation in Parspat
Parses are constructed on-line and bottom-up. Ambiguous parses are shared as much as possible. Parses can be inspected at any moment.

e. Semantics
Reconstruction of all possible derivations.

f. examples
see section 2.2.1.

2.2.8.3 Output-variables

a. application and semantics
With variables all kind of structures can be built. They can be brought to the output string. If during recognition, parsing or transduction no rhs is applicable for a character it is brought to the output string. This happens as soon as possible. In the case of a phrase-structure grammar the output string will contain (after successful recognition) the start symbol, accompanied by its variables. These variables are also brought to the output string, in the form of pairs (variable, value). In the case of ambiguities all sets of pairs will be written. With transduction the same thing happens, but then more symbols can appear in the output, together with their associated variables. Cascaded grammars are treated as a whole : an output string will become visible when it is not denoted as an input string for some other grammar.

f. example
S(O:result) :: NP(voud,time) { result := 'NP:' || time || voud }, [VP(voud,time)] .
! in the output string the variable result will appear together with its value, which consists of the concatenation of the string 'NP:' with the values of the parameters time and voud, which are returned by the nonterminal NP. !

2.2.8.4 External tree building

a. application
Intermediate texts, like structured text-corpora , often have the form of a labeled tree. As we explained with tree-matching, it is possible that during a match a skip has to be performed to the end of the current level in the tree. This can be optimized by placing a pointer at the start of the level. The construction of such intermediate tree-structured texts can be performed by the use of the "build"-operator. As regards on-line behaviour the same remarks are valid as stated under reports.

b. functioning within formalism
As an action.

c. lexical considerations
The '(' and ')' are the reserved tree symbols.

d. syntax
BUILD_DECLARATION:: B , ': ', ['(' | ')' | '%' | VARIABLE_EXPR]+ .

f. example

A :: * {B: (% }, b .

The "build"-operator B will start to construct a new substructure (denoted by the '(') with as the first character the last read terminal.

2.2.8.5 Transduction

a. application

Transduction by general rewriting was mentioned in phonological rewriting and machine translation.

b. functioning within formalism

All transduction rewrite rules will operate in parallel. Transduction stops when no more rules are applicable. This process may give rise to a number of ambiguous transductions. In this section we provide for a number of shorthands and extensions which proved to be useful in a number of applications and which adhered to the formalism the grammar writer had in mind. It is possible that in the course of evolution more shorthands will be developed.

c. lexical considerations

Cover symbols.

Nonterminals may be suffixed by the "cover symbol" '^'. During parsing a covered nonterminal will be replaced by the terminals which are at the leaves of its associated parse tree(s). The current scope for covering at the lhs is the governing cs rule. The current scope for covering at the rhs is the current regular expression. If the notion is not known then the scope is widened to the surrounding regular expression, etcetera.

A further suffixing is allowed in order to reference not the leaves of the parse tree but some symbol within the parse tree, eventually suffixed again by the cover symbol.

Shorthands.

Shorthand1 : suffixed notions.

Nonterminals may be suffixed by an integer. If T is a nonterminal then the use of "T1" implies the existence of the rule "T1 :: T".

Shorthand 2 : reference of notions.

Sequences of symbols can be denoted by a nonterminal with a cover symbol. With regular expressions this is not possible. We therefore introduce the assignment operator '$' (like in Snobol) which assigns the value of the last regular expression (in the most simple case : a notion) to a variable, which may be an alphanumeric name. This is essentially the same as rewriting this variable to that regular expression in a separate rule. For instance, the rule

a, $1, b :: b, [a, =, b]*$1

is equivalent with the two rules a, V1^, b :: b, V1 . V1 :: [a, =, b]* .

In essence a nonterminal functions as a description of a sequence of terminals. After transduction it disappears. The coverage of a nonterminal may be partially rewritten. In that case we may reference it by the same name, but it has to be followed, between labeled brackets, by the rewritten coverage. "Arbs" and "lines" are allowed. Examples :

α, A1:($1,B1^,$2,C1^,$3), β :: β, A1:(-$1,C1,-$2,B1,-$3), α.

Within the labeled bracket of A1 a reordering is indicated.

For simple applications a shorthand is allowed : when the sequence of arbs and/or lines in the lhs is the same as at the rhs, and there are the same number of arbs, then assignment to variables is not necessary. The above example can then be transformed into the following, simplified, one:

```
α, A1:(-,B1^,-,C1^,-), β :: β, A1:(-,C1,-,B1,-), α.
```

If a symbol in the lhs is not a cover symbol then it may be rewritten. When during rewriting a cover symbol is used then its value is taken from the rhs. The above example could also have been written as :

```
α, A2, β :: β, A1, α.
A1 :: (-,C1,-,B1,-).
A2 :: A1:(-,B1^,-,C1^,-).
```

(The repetition of A1 as a label is only necessary to get a reference for the arbs).

Shorthand 3 : Referencing of the choice between alternatives.

The choice of an alternative in the rhs may be referred to in the lhs by an alphanumeric label.
Example :
$\alpha_1\ [\ 1\!:\ \beta_2\ |\ 2 : \gamma_2\ ]\ \delta_1\ ::\ \alpha_1\ [\ 1\!:\ \beta_1\ |\ 2 : \gamma_1\ ]\ \delta_1$.

<u>d. syntax</u>

```
NOTION :: USER_NOTION, '^' .
```

<u>d1. Restrictions in Parspat</u>
Cover symbols are implemented at the moment of writing, but not the other shorthands. We therefore leave out the syntax for these shorthands.

<u>e. semantics</u>
Heuristics, stemming from the practice of phonological rewriting, may be activated to reduce the number of ambiguities:
- if 2 rhs's are applicable, the longest one is chosen
- rewriting according to a rule is performed as soon as possible; the piece of text which was matched by the rhs of the rule is no longer subject to other transformations.

<u>f. examples</u>

```
1       A^, B :: B, A
        A :: a, C
        B :: b
        C :: c
```

! The input 'bac' will be rewritten as 'acB'. !

```
2       E,SG^ :: e,SG.
        SG :: CONS , CONS  & `[k,l].
        CONS :: `[ a | e | i | o | u ]1 .
```

! An 'e' followed by a string of 2 consonants which is not 'kl' will be rewritten to 'E'. !

```
3       A^[2,1], b :: A .
        A :: B, C | C, B .
        B :: P, q | p, s .
        P :: p, t .
        C :: q | r, s.
```

! With the input 'rsptq' the nonterminal A in the first rule will get a parse tree C: (r , s) B: (P: (p , t) , q). The notion A^[2,1] means : get the 2nd symbol on the level 1 below A, which is B; get from that B the 1st symbol on the level below B, which is 'P'. The output of this transduction grammar will therefore be : 'Pb'. If we had written A^[2,1]^ then this would have been evaluated to P^, which denotes the terminal symbols below P. The output of the grammar would have been : 'ptb' . !

```
4       A^ < A , A^ .
        A :: a..z.
```

With this transduction grammar all double lower case characters will be singled.

```
5       $3, $1, $4, $2 :: [a]1$1, [b]*$2, A$3, [g]+$4.
```

Three regular expressions and a nonterminal are rewritten. Note that in the lhs '[$3]' could have been written also as 'A^'.

2.3 Relation to other formalisms

In this section we will relate the main existing methods for the formulation of syntactic patterns to the unifying formalism. A broad division is made between procedural and non-procedural interpretations.

2.3.1 Non-procedural recognition and parsing

2.3.1.1 String- and tree-matching

There are a large number of applications which make use of string- and tree-matching. In Corpus Linguistics corpora are available with texts which are organized as strings or as labeled trees of potentially unlimited size. The tree-structure concerns for instance the structure volume-chapter-paragraph-sentence, with appropriate labels, or a parse tree (see also the examples in chapter 8).
In chapter 1 we discussed the on-line aspects of the matching process. There we mentioned the "Linguistic String Parser" (Sager, 1981) which makes use of restriction rules which operate on a parse tree. The restriction rules, which can become very elaborate, are written in a separate language. It is possible to rewrite these rules in the unifying formalism, as is demonstrated in (De Jong and Masereeuw, 1987).
The restriction rules are formulated in terms of routines for navigating in the parse tree. Some routines are predefined. Other routines can be defined using these basic routines. There is for instance a routine for going from a node to a parent node, or for descending or ascending until the test for a node with some property fails or succeeds.
We show a fragment of a much simplified version of the LSP grammar.

Context-free Component:

```
        <ASSERTION>     ::= <SUBJECT> <VERB> <OBJECT>.
        <SUBJECT>       ::= <NSTG> / $NULLWH.
        <OBJECT>        ::= <NSTG> / $NULLOBJ.
        <NSTG>          ::= <LN> $N <RN>.
        <RN>            ::= <WHSTG> / (alternatives) .
        <WHSTG>         ::= $WH <ASSERTION>.
```

Routines:

```
        IMMEDIATE (X) = ASCEND TO X.
```

Restrictions:

```
        DZERON1 = IN SUBJECT RE $NULLWH: IMMEDIATE WHSTG
                  OF IMMEDIATE ASSERTION EXISTS.
        DZERON2 = IN SUBJECT RE NSTG: IT IS NOT THE CASE
```

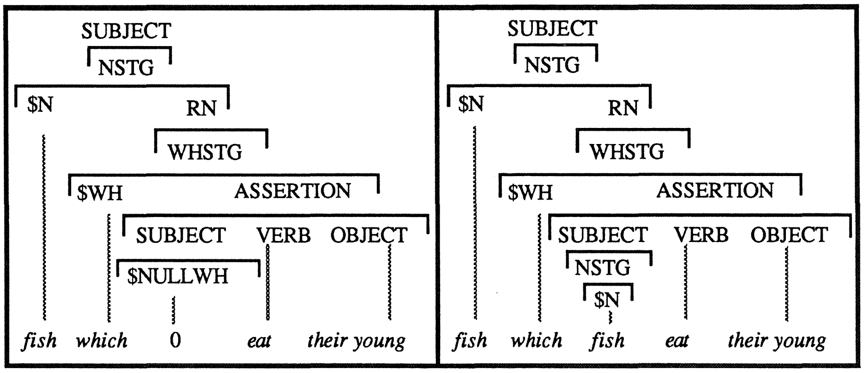
THAT IMMEDIATE WHSTG OF IMMEDIATE
ASSERTION EXISTS.

Test sentences (only the Subject phrases shown):

(a) Fish which eat their young ...

(b) *Fish which fish eat their young ...

The first sentence is grammatical, the second is an ungrammatical one. A parser for the Context-free Component will produce the following parses.



That a parse is also obtained for the ungrammatical b-sentence is due to the Subject which may be rewritten as either zero or a full NP. There is nothing in the CF rules that prevents a full NP in that position. Instead, that is the job of the restrictions, in particular restriction DZERON2.
Both of the restrictions shown above use a routine IMMEDIATE which says "go up one level to a node specified through parameter X". Restriction DZERON1 states that a parse such as for sentence (a) may pass: a Subject dominating a node $NULLWH must have a node ASSERTION above itself and above that there should be a node WHSTG. This restriction actually serves to rule out zero-Subjects in other environments than these. Restriction DZERON2 is the one that actually rules out the parse in the right figure: it says that if you have a subject NSTG, it should *not* have nodes ASSERTION and WHSTG above it, as it has in the parse in the right figure.

In the unifying formalism the grammar can be written as follows.

Context-free Component:

```
ASSERTION :: SUBJECT, VERB, OBJECT.
SUBJECT   :: NSTG | $NULLWH.
OBJECT    :: NSTG | $NULLOBJ.
NSTG      :: LN, $N, RN.
RN        :: WHSTG | (alternatives) .
WHSTG     :: $WH, ASSERTION.
```

Routine:

```
IMMEDIATE(C:y,C:x} :: (..( {C:y}, *, :( {C:x}.
! Routine IMMEDIATE accepts two cooperation parameters x and y, and states that y
should be exactly one level above position x. !
```

Restrictions:

```
DZERON1 ::
  (..(, SUBJECT{c:a}:($NULLWH), )..)
! Somewhere in the tree there is a Subject with a $NULLWH node under it. !
&
  (..(, ASSERTION{c:b}, )..)
! and somewhere else there is a node ASSERTION !
&
 IMMEDIATE(C:a, C:b)
! and a relation IMMEDIATE exists between them. !
&
 (..(, WHSTG{C:c}, )..)
&
 IMMEDIATE(C:a, C:c}
|
  (..(, `[ SUBJECT:( $NULLWH ) ].
```

(Restriction DZERON2 can be translated along the same lines.)

The restriction DZERON1 should then be read as follows:
- you have to have a SUBJECT node dominating a $NULLWH node; its position is passed on through cooperation a.
- you have to have an ASSERTION node at position b.
- there should be a relation IMMEDIATE between a and b (Note that)..) is the converse of (..(and makes these coordinated rules return to the same level).
- moreover, there should be a node WHSTG at position c, such that there is a relation IMMEDIATE between b and c.
- the last line says that if these conditions are not met, there should not be a SUBJECT dominating $NULLWH at all.

2.3.1.2 Attribute, affix and augmented phrase structure grammars

An attribute grammar associates a set of attributes with each symbol $X \in N$. Each attribute represents a specific (context-sensitive) property of the symbol X and can take on a specified set of values. The mechanism is the same as in programming languages with procedures provided with formal parameters. These parameters may be assigned values and may be compared with each other.
The formalism of affix-grammars (Koster, 1970) originated at the same time as attribute-grammars. They do not differ in an essential way.
The same formalism is known in computational linguistics as augmented phrase structure grammar . Winograd(1983) : "Each system based on augmented phrase structure grammar provides a parsing strategy and a particular set of augmenting mechanisms that are compatible with that strategy. Therefore, although they are all based on the same principles, they differ in the detailed nature of conditions and actions and in their efficiency and convenience". A linguistic application is for instance the coordination of number and time between noun-phrase and verb-phrase (Oostdijk, 1984).
An example of an attribute grammar, written as a U-grammar, has already been given in section 2.2.5.

2.3.1.3 Transformational grammars

Tree grammars operate on the tree-datastructure. In transformational grammar this tree is implicitly formed by a rewriting with context-free rules. The nodes in the tree are labeled by symbols of the vocabulary of the grammar and are provided with "distinctive features" which originate from the lexicon.
Transformations are expressed in one of the following operations on the parse tree :
- insertion of branches and/or features,
- deletion of branches,
- movement, seen as a combination of insertions and deletion,
- change of features.

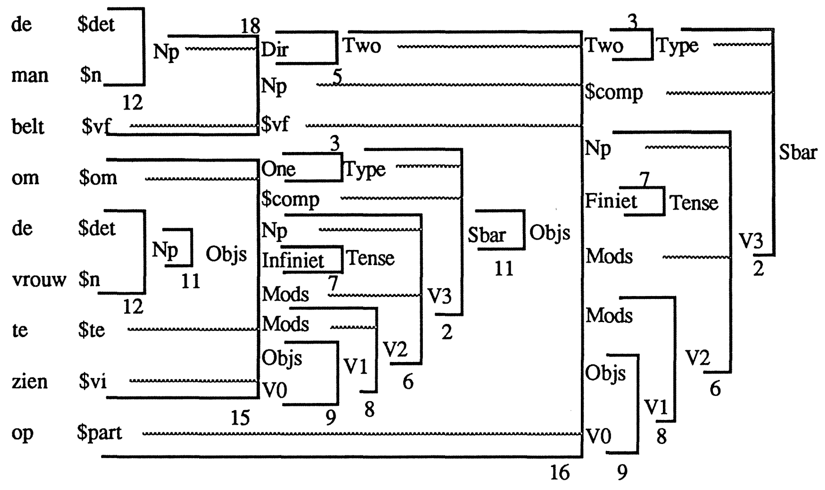
In linguistic theory (Chomsky, 1965) transformations may be optional or obligatory. The current trend is to make all transformations optional and to reduce them to only one transformation, "move alpha", which is a variable over all syntactic categories. Usually an ordering is assumed about the applicability.
After the initial formulation of the theory it gradually changed its appearance. In the so-called "trace-theory" markers are assigned to places where categories disappeared because of a "move alpha" application. The trace is a relation between the old and the new position in order to make the move recoverable. A number of restriction rules have to restrict the resulting overgeneration.
We are not concerned with the ongoing efforts in this respect but only with the effects they have on the formalisms in which (pieces of) transformational grammars are written.
We present as an example a piece of a transformational grammar according to some EST principles, written in the unifying formalism. It was provided by J. van den Hoek. The drawing of one of the parses is based upon the output of the Parspat system.

Grammar :

```
2     Sbar   ::    Type, $comp, V3.
3     Type   ::    One | Two.
4     One    ::    Indir .
5     Two    ::    Dir | Qu.
6     V3     ::    Np, Tense, Mods, V2 .
7     Tense  ::    Finiet | Infiniet .
8     V2     ::    Mods, V1 .
9     V1     ::    Objs, V0 .
10    Mods   ::    [ Pp | Sbar ]* .
11    Objs   ::    [ Np | Pp | Sbar ]* .
12    Np     ::    [$det], [$adj]*, $n .
13    Pp     ::    $p, Np .
14    One, $comp, Np, Finiet, Mods, Mods, Objs, V0
                   ::    $dat, Np, Mods, Mods, Objs, [$part], $vf .
15    One, $comp, Np, Infiniet, Mods, Mods, Objs, V0
                   ::    [ $om ], Mods, Mods, Objs, [$part], $te, $vi .
16    Two, $comp, Np, Finiet, Mods, Mods, Objs, V0
                   ::    Two, Np, $vf, Mods, Mods, Objs, [$part] .
17    Dir, Np, $vf ::    Np, $vf .
18    Qu, Np, $vf  ::    $vf, Np .
```

Sentence :
"de man belt om de vrouw te zien op"

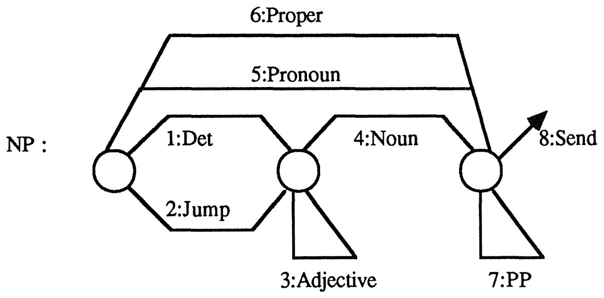
Parse :



2.3.1.4 Augmented Transition Networks

Augmented Transition Networks are at present the most widely used method for analyzing natural languages. They are derived from finite state automata. An ATN consists of a collection of labeled states and arcs. States are connected with each other by arcs creating a directed graph. For each nonterminal symbol there is a separate graph. Jumps to and returns from graphs are performed by the instructions PUSH and POP. To make the model more powerful each arc is equipped with a test and a sequence of actions. The test is an arbitrary condition which must be satisfied before the arc can be traversed. Actions are executed during the transition through the arc. Tests may examine the contents of registers and actions may assign arbitrary values to them.
ATN's can be written in the unifying formalism in a straightforward manner by using the cf sub-formalism with regular expressions and variables within actions. We illustrate this by a simple example given by (Winograd, 1983, page 215). It concerns the phenomenon of number agreement in a NP Network.

Example written in original formalism :



Conditions and Actions on the arcs (* denotes the entry on the arc) :
1: Action: set Number to the Number of *
4: Condition: Number is empty or Number is the Number of *

```
   Action: set Number to the Number of *
5: Action: set Number to the Number of *
6: Action: set Number to the Number of *
```

Example rewritten in unifying formalism :

```
NP ::  [ [ Det (Number) ] ,
         [ Adjective ]* ,
           Noun (Number1) { Number = '' | Number1 & Number := Number1 } ,
       | Pronoun (Number)
       | Proper (Number)
       ]1 ,
         [ PP ]* .
```

2.3.2 Non-procedural Transduction

2.3.2.1 Syntax-directed Translation Schemata

In syntax-directed translation schemata (Aho and Ullman, 1972) the rules of a cfg are enriched by a transduction-component. I.e.

```
        if the grammar contains the rules:
                S :: NP V , (transduction:) V NP
                NP :: DET ADJ N , (transduction:) N ADJ
        and the lexicon contains: (VP : works; N : man ; Det : the ; ADJ : old)
        and the input is : 'the old man works'
        then after recognition of the second rule NP will be associated with 'man old'
        and after recognition of the first rule S will be associated with 'works man old'
        which transduction will be the output .
```

Input and output are strictly separated; an intermediary associated string will never be subject to a new analysis or a new transduction.
We are not aware of any substantial use of this formalism. However, it combines transduction and recognition according to a grammar, a feature which is absent from most other formalisms.

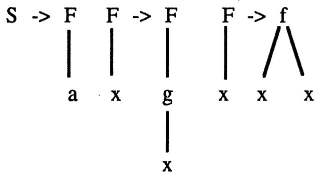
Example rewritten in the unifying formalism :

```
        S:(V, NP(-)) ::        NP(-), $V.
        NP:(N,Adj) ::          $det, $adj, $N.
```

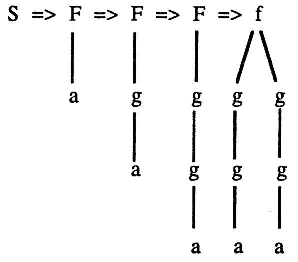
2.3.2.2 Web-, Graph- and Tree Grammars

In section 2.3.1.1 we discussed the matching of tree-structured files. Pattern matching in trees also occurs frequently in the context of tree replacement systems and has applications in different areas of Computer Science, including automatic implementation of abstract data types, code optimization, automatic proof systems, syntax-directed compilation and evaluators for programming languages such as LISP.
Strings can be generalized to trees by allowing the concatenation operator to be multidimensional. Tree grammars specify how to replace subtrees by other subtrees. If trees are extended to graphs we may imagine graph-grammars which direct the replacement of sub-graphs by other sub-graphs. In the general case extra directions are needed to guide the attachment of a sub-graph because the edges of the replacing sub-graph may be different.

Conditions on the replacement may introduce context-sensitivity. The reader is referred to (Ehrig e.a., 1983) for an overview and a bibliography.
For applications in SPR which follow the linguistic approach the formalism may usually be simplified. The structures one deals with are usually trees. Subtrees which have to be replaced contain the end-leaves so that no problems arise with the redirection of the connection of edges.
We take an example from (Schimpf and Gallier, 1985, page 29). They give a context-free tree grammar (CFTG) with production rules :



Here 'a' and 'g' are terminals, 'x' stands for any node with underlying tree. A sample derivation is as follows :



The example can be rewritten in the unifying formalism as:

```
S :: F , '(' , a , ')' .
F , '(' , X^ , ')'  ::  F , '(' , g , '(' , X , ')' , ')' .
F , '(' , X^ , ')'  ::  f , '(' , X , X^, ')' .
X :: a..z , [ '(' , Y, ')' ] .          ! these two rules describe a node with !
Y :: Y , Y | X .                        !  any underlying tree                  !
```

2.3.2.3 Formula Manipulation Systems

Formula manipulation systems have a long tradition. Knuth (1968, vol. 1, page 337) gives examples on the computational treatment of symbolic differentiation. Dershowitz (1985) redefines the rules for symbolic differentiation as general rewrite rules and studies the formal properties of general rewrite systems. We repeat some of the grammar rules for symbolic differentiation :

$D_x\, x : 1$.

$D_x\, \alpha : 0$.

$D_x\, (\alpha + \beta) : D_x\, \alpha + D_x\, \beta$.

$D_x\, (-\alpha) : - D_x\, \alpha$.

$D_x\, (\alpha\, \beta) : \beta\, D_x\, \alpha + \alpha\, D_x\, \beta$.

where D_x is the differentiation operator and 'a' stands for any constant symbol other than x. α and β are variables of the rewrite system and match any term, while x is a constant of the system and matches only itself.
The rules can be rewritten in the unifying formalism as follows :

```
! differentation !
1 < D, Var .
0 < D, Constant .
D^, Expr1^, [1:'+' | 2:'-']1, D^, Expr2^ :: D, '(', Expr1, [1:'+' | 2:'-']1, Expr2, ')' .
'-', D^, Expr^ : D, '(', '-', Expr, ')' .

! distribution !
Expr2^, D^, Expr1^, '+', Expr1^, D^, Expr2^ :: D, '(', Expr1, '*',  Expr2, ')' .

! simplifications !
0 < Expr, '*', 0.
0 < 0, '*', Expr.
Expr^ < 0, [ '+' | '-' ]1, Expr.
Expr^ < Expr, [ '+' | '-' ]1, 0 .

!  arithmetic expression !
Expr :: E .
E :: ['-'], T, [ ['+' | '-' ]1, T] .
T :: F, [ [ '*' | '/' ]1, F]1 .
F :: Varconst | [D], '(', E, ')' .

! lexical symbols !
Varconst :: Var | Constant .
Var :: x .
Constant :: a..g | 1..9 .
```

On the basis of these rules the Parspat system performs symbolic differentiation. More rules can be added for division etc. and for more simplification.

2.3.3 Comparison with special programming languages for pattern recognition

2.3.3.1 Comit and Meteor

In the programming languages Comit and Meteor statements are roughly in the form of general rewriting-rules. The lhs consists of a pattern with one or more variables. In case of a match of the lhs with the input to this rule (generally a string) the rhs indicates the value which has to be assigned to the variables in the lhs. A variable may have an attribute attached to it. This attribute may be assigned a value too like, for instance, the predicate "noun". The attributes may be used in the formulation of the patterns.
The programmer has to specify the order of execution of the rules.

Example written in original formalism :

($1/V $ to $1/P) (1 4 2) ! a string, consisting of a verb (1 word: "$1", with category V), which is implicitly assigned number 1, followed by an arbitrary number of words ("$"), implicitly assigned number 2, followed by "to", implicitly assigned number 3, followed by a Personal Noun, implicitly assigned number 4, is rewritten as a string consisting of the 1st, the 4th and the 2nd word. For instance, the string "handed the bottle to Andy" is transformed into "handed Andy the bottle". We left out the control part. !

Example rewritten in unifying formalism :

```
var1, var4, var2 :: $V(var1), - {var2 := var2 ||%}, 'to', $P(var4) .
```

2.3.3.2 Snobol

Snobol looks like Meteor. The control structure is of the same primitive kind: as in programmed grammars each program-rule has a success- and a failure-field.
Because of the large number of pattern matching facilities it is a popular language among linguists. In the patterns one may use nonterminals which may be defined in further rules (even recursively). It is therefore straightforward to write a cfg in Snobol. Problems arise with ambiguities: the system can only handle backtracking within one program-rule.
Pattern-variables are evaluated in run-time. This contributes to the power of the language but, at the same time, slows down the runtime-system.
Recognizers for (non left-recursive) context-free grammars may be written in a recursive descent fashion, using these pattern-variables.
We give some examples of the pattern matching functions of Snobol because of the attraction they have for linguistic users and present their counterpart in the unifying formalism.

Snobol	Unifying formalism
ARB	=
not provided in Snobol: to match everything, even the remainder of the pattern, up till the end of the current tree-level (enables the recognition of all occurrences of the remaining pattern)	-
ARBNO(S)	[S]+
LEN(5)	[*]5
BAL	(-)
SPAN('pq')	[p | q]*
BREAK('pq')	`[p | q]*
ANY('pq')	[p | q]1
NOTANY('pq')	`[p | q]1
Q ABORT | P	P & Q`

2.3.3.3 Icon and Summer

The control structure of Snobol is a primitive one. Its designer, Griswold, therefore developed a successor to Snobol, called Icon (Griswold and Griswold, 1983). This language has the normal control sequences and procedure mechanism of higher level languages like Pascal. A novel concept is the "generator": an expression which may yield more than one result (for instance, all the matching points of a substring in a string). Generators may be used as parameters for other generators. The results may be handled in an iterative way (using all possible results) or in a goal-directive way: the evaluation mechanism then attempts to produce at least one result for all the expressions involved in the evaluation.
The implicit backtracking process may be influenced by intermediate statements. Icon has no implicit facility for backtracking of data.
The pattern matching facilities are more or less the same as in Snobol.

At the same time Klint (1985) developed the language "Summer". He made backtracking explicit by the concept of "recovery blocks" which may be "tried", like the alternatives in a grammar, but in some sequence. Such a block is syntactically structured like a block in an algol-like language. The data-objects may each maintain their own "cursor", influenced by current pattern matching, and the programmer may make use of these cursors.

The string pattern matching functions are about the same as in Snobol. The recognition-strategy makes use of backtracking, which may be influenced by the programmer. Klint (1982, p. 17) writes: "In general, it might be a better idea to make the recognition strategy invisible at the programming language level and to let the implementation choose the best strategy for a given problem. This line of development is interesting but falls outside the scope of the current work".
The recognition strategy of the Parspat system is completely hidden to the user.

2.3.3.4 Prolog

Like Icon, Prolog (Clocksin and Mellish,1981) is a goal-oriented language, but oriented more towards the retrieval of relations. Like Snobol, it is a simple thing to write cfg's in Prolog in a recursive descent fashion (e.g. Dahl, 1985). The nonterminals, in the form of relations, may be attributed by parameters. In such a way it is possible to write attribute grammars with the facility of unification of variables. The facility of automatic backtracking provides for the handling of ambiguities. The price is that recognition may cost exponential time. Evaluation is "depth first". The programmer has to realize this manner of evaluation and may influence the process of backtracking. The ordering of the rules is important.

In comparing the syntax of Prolog with the unifying formalism the former seems almost to be a subset of the latter. In the unifying formalism more facilities are available for the matching and assignment to variables, as well as for the handling of Boolean negations.

A grammar in the unifying formalism is a set of rules in which the ordering is not important. There is nothing like a control structure, a procedure or a function. Only rewritings of rhs's are given. All conditions that are to be fulfilled in order to select the rewriting are present in the rhs. Each rule stands on its own and will be applied each time when its rhs is matched. Repetitions will therefore be performed automatically. Constructions like "while A do B" and "do for all a in A" are obsolete.

The "if-then-else" and "case" construction are handled by the alternative-notation. Two sequence constructs exist : one within an alternative and one within an action-part. No side-effects are possible, because each rule stands on its own and has no interaction with other rules. Each variable is local to a rule. Failure of a derivation implies the deletion of everything that belongs to that derivation.

2.4 Useful effects of combination of formalisms

In the former section we related the unifying formalism to other, already existing formalisms by making combinations of some sub-formalisms. It seems possible also to identify new applications by making other combinations.
The combination of tree pattern matching and grammatical description gives rise to the application of information retrieval in free text database systems.

Unification of Grammars and Description- and Query-Languages for Textual Data bases.

Testing of hypotheses on linguistic data bases may give rise to the formulation of complicated queries. It is often difficult to foresee beforehand what kind of queries will be put to such a data base system. One does not know beforehand the relations that will be needed. In these cases a relational data base system cannot be used, simply because we don't know which relations to store. The other choice is to use a hierarchical data base system.

The advantage of a relational model may be the simple use of such operators as "selection", "projection" and "join", well-known from relational algebra. The last-mentioned operator introduces some kind of intelligent combination of stored relations that is absent from hierarchical models.

The question arises how to put more intelligent queries to a hierarchically conceived data base which contains free text (where "free" means that for every record there exists, in principle, no upper bound on its length).

There is a simple link between syntactic recognition and retrieval in a hierarchically structured data base. In the latter case we formulate queries in which the sequence of vertical components in the tree is important. These vertical sequences are mostly of a simple nature and are formulated by procedural query-languages in which one navigates in a horizontal and vertical direction through the tree-structure. However, the expressive power of the queries can be increased considerably when we allow for the full power of grammatical notation on the vertical lines.

Moreover, we will have the advantage of a concise, elegant notation and the simple semantics of the description of formal languages where a grammar describes a set of sentences.
By allowing grammar-notations on the horizontal lines we introduce the possibility of linguistic descriptions on free formatted text.

The Parspat system may act, on one extreme, as a hierarchical free-text data base system and, on the other extreme, as a grammatical parser. In between it is a general tool in which we may organize and query information written in natural language and structured according to the users' wishes. This structure takes the form of a labeled tree where each subtree may differ in structure from another sub-tree.

A formal grammar describes a language. This language consists of a finite or infinite number of sentences. An existing data base may be regarded as such a sentence. The description of the form of that data base, normally written in what is called the "Data Description Language" (DDL), may be given by a formal grammar.

We may look at queries as the description of a language, formed by all the possible answers. Formulated as pattern grammars they describe a (possibly infinite) set of sentences, as does the language generated by that grammar. The answering of a query may then be looked upon as the recognition of the data base which acts as the input-sentence to the pattern grammar. Ambiguity during recognition accounts for the possibility of two or more answers.

We will clarify our approach with a simple example, taken from a standard textbook on data base systems (Date, 1981), which demonstrates the standard operations: union, intersect, minus, select, project and join for a hierarchical data base system. In each case we will show the corresponding grammar-notation.

Date uses one example consistently throughout his text-book. It concerns an education database of a commercial firm in which both teachers and students are employees. The courses are defined by course#, title and description. Each course has, possibly, prerequisite courses and is offered on a number of dates and locations. It has a teacher and students, presented with their names; in addition, students have grades (for figures see Date pp. 280, 281).

The structure of this data base may be described by a formal grammar as a Data Description Language. We choose for it an ecfg with tree symbols, as a subset of the unifying formalism.

The grammar for the education database is divided into two parts. These parts resemble the distinction which is made in existing data base systems between different levels of DDL's. The first part describes the logical structure, the second part the physical structure of the data base (the two parts are not treated in a different way: they are mixed during compilation).

The grammar becomes:

```
Data_Base    :: ['Course':(Course)]+.
Course       :: Course#, Title, Description, ['Prereq':(Prereq)]*, ['Offering':(Offering)]+.
Prereq       :: Course#, Title.
Offering     :: Date, Location, Format, 'Teacher':(Teacher), ['Student':(Student)]+.
Teacher      :: Emp#, Name.
Student      :: Emp#, Name, Grade.
------------------------------------
Course#      :: number.
Title        :: string.
Description  :: string.
Date         :: date.
Location     :: string.
Format       :: string.
Emp#         :: number.
Name         :: string.
Grade        :: number.
number       :: [1..9]*, [0..9].
string       :: [a..z | A..Z]+.
```

The labels between quotes are not strictly necessary. They act as markers in a variable length environment and label sub-trees.
As the Data Manipulation Language we use pattern grammars. For them we choose the sub-formalisms of Ecfg's (but not recursive), tree symbols, don't cares, lines, booleans, reports and variables. The variables make possible, in a data base environment, the operation "join".

We now present the basic retrieving functions in terms of our formalism :

UNION : retrieve "all Course#'s for courses which either have as a prerequisite
Course#=10 or which are located in Stockholm" :

S :: -, Course:(* {R:1}, -, [Prereq:(*, 10, -) | Offering:(*, Stockholm, -)], -).

INTERSECT : retrieve "all Course#'s for courses which have as a prerequisite
Course#=10 and which are located in Stockholm"(straightforward
syntactic variation of UNION) :

S :: -, Course:(* {R:1}, -, Prereq:(*, 10, -), -, Offering:(*, Stockholm, -), -).

MINUS : retrieve "all Course#'s for courses which have as a prerequisite
Course#=10 and which are not located in Stockholm" (straightforward
syntactic variation of INTERSECT) :

S :: -, Course:(* {R:1}, -, Prereq:(*, 10, -), -, Offering:(*, Stockholm`, -), -).

SELECT : retrieve "all Course#'s for courses in Amsterdam" :

S :: -, Course:(* {R:1}, -, Offering:(*, Amsterdam, -), -).

PROJECT: retrieve "all Emp# of all students" :

```
S :: -, Course:(-, Offering:(-, Student:(* {R:1}, -), -), -).
```

JOIN (with PROJECT) : retrieve "all Course#'s for courses where the student is the teacher of one of the prerequisite courses" :

```
S :: -, Course:(S1(c1,t1) & S2(c2,t2) {t1=t2 & c1=c2}).
S1(c1,t1) :: * {R:1}, -, Prereq:(* {c1 := %}, -), -, Offering:(-, Student:(* {t1 := %}, -), -).
S2(c2,t2) :: * {c2 := %}, -, Offering:(-, Teacher:( * {t2 := %}, -), -).
```

In the last query the variables c1, c2, t1 and t2 are used.

2.5 Syntax of the formalism

All rules from the preceding sections are collected below.

```
2.2.1 Basic formalism
GRAMMAR::                 [ RULE ]+.
RULE::                    ALTERNATIVES , ['::' | < ]1, [ ALTERNATIVES ], '.' .
ALTERNATIVES::            ALTERNATIVE, [ '|' , ALTERNATIVE]*.
ALTERNATIVE::             UNIT, [ ',' , UNIT]* .
UNIT::                    NOTION .
NOTION::                  NONTERMINAL_SYMBOL | INTERMEDIATE_SYMBOL |
                           TERMINAL_SYMBOL.
NONTERMINAL_SYMBOL::          ALFANUMS.
INTERMEDIATE_SYMBOL::         ALFANUM.
TERMINAL_SYMBOL::             BASIC_SYMBOL.
BASIC_SYMBOL::            ALFANUM  |  '<' , [ OCT_DIGIT  ]+ , '>' .
----- end 2.2.1

2.2.2 Regular expressions.
UNIT::                    ENCLOSEDPART .
ENCLOSEDPART::            '[' , ALTERNATIVES , ']', [ '1' |  '*' | '+'].
---- end 2.2.2

2.2.3.1+2+3  Don't care, arb, line
TERMINAL_SYMBOL::    '*' | '=' | '-' .
--- end 2.2.3.1+2+3

2.2.3.4 Actions
NOTION::          USER_NOTION .
USER_NOTION::     RANGE_SYMBOL, '..',  RANGE_SYMBOL .
---- end 2.2.3.4

2.2.4
ENCLOSEDPART::  '['  , ALTERNATIVES , ']',
                [ ACTIONS], [ DEC_DIGIT |  '*' | '+'], [  ACTIONS ] .
ALTERNATIVE::   UNIT, [ACTIONS], [ ',', UNIT, [ACTIONS] ]* .
ACTIONS::       '{', action ,  ['&' ,  ACTION ]*, '}'.
ACTION::        REPORT_DECLARATION | COOP_DECLARATION |
                        ASSIGNMENT | TEST | BUILD_DECLARATION |
                        GRAMMAR_CALL | PROCEDURE_CALL .
--- end 2.2.4

2.2.5 Variables
NONTERMINAL_SYMBOL::       ALFANUMS, PARAMETERLIST.
PARAMETERLIST::          '({', F_OR_A_PARAMETERS , '})' |
                         '({' , ACTUAL_PARAMETERS , '})' .
F_OR_A_PARAMETERS::  PARAMETER_DECL , [',', PARAMETER_DECL]* .
ACTUAL_PARAMETERS::ACTUAL_NAME , [',', ACTUAL_NAME ]*.
PARAMETER_DECL::         [ I | O | I , O ]1, ':' , ALFANUMS .
ACTUAL_NAME::            ALFANUMS .
ASSIGNMENT::             ALFANUMS , ':=' , VARIABLE_EXPR .
TEST::                   VARIABLE_EXPR , [ '=' | '/=' ]1 , VARIABLE_EXPR .
VARIABLE_EXPR::          VARLIT , [ '||' , VARLIT]*.
VARLIT::                 ALFANUMS | LITERAL | '%' .
GRAMMAR_CALL::           ALFANUMS, '(', [ ALFANUMS, ',']6, ALFANUMS , ')' .
PROCEDURE_CALL::         ALFANUMS, [ '(', ALFANUMS, [ ',',
                          ALFANUMS]* , ')' ].
LITERAL::                 "' , = , "' .
--- end 2.2.5

2.2.6.1 Cooperation
ALTERNATIVES::           ALTERNATIVE, [ '&' , ALTERNATIVE]*.
PARAMETER_DECL::         COOP_DECLARATION .
COOP_DECLARATION::  C, ':', ALFANUMS .
--- end 2.2.6.1

2.2.6.2 Boolean negation
ENCLOSEDPART::  '`[' , ALTERNATIVES , ']',
                     [ ACTIONS], [ '1' | '*' | '+'], [ ACTIONS ] .
NOTION::             [ NONTERMINAL_SYMBOL | INTERMEDIATE_SYMBOL |
                      TERMINAL_SYMBOL ]1, '`'.
--- end 2.2.6.2

2.2.7.1 Tree symbols
UNIT::                TREEUNIT | TREESUNIT.
TREEUNIT::           [ '(' | NOTION, ':(']1,  ALTERNATIVES, ')', [ ACTIONS ].
TREESUNIT::          [ '(..(' | NOTION, ':(..(']1,  ALTERNATIVES, ')..)',
                      [ ACTIONS ].
--- end 2.2.7.1

2.2.7.2 Lexicons
TERMINAL_SYMBOL::        "$", ALFANUMS.
action::                 'update(', LEXICON_STRING, [ LEXICON_NAME ], ')' |
                         'delete(', LEXICON_STRING, [ LEXICON_NAME ], ')' .
LEXICON_STRING::          VARIABLE_EXPR, [ '#', VARIABLE_EXPR]* .
--- end 2.2.7.2
```

```
2.2.8.1  Reports
REPORT_DECLARATION::          [R | S]1 , ':' , [ DEC_DIGIT  ] + .
--- end 2.2.8.1

2.2.8.4  Builds
BUILD_DECLARATION::  B , ': ', [ '('  | ')'  | '%' | VARIABLE_EXPR ]+ .
--- end 2.2.8.4

2.2.8.5  Transduction
NOTION::             NONTERMINAL_SYMBOL, '^' .
--- end 2.2.8.5

2.2
ALFANUM::            DEC_DIGIT | CHAR | "', <000>...<377>, "' .
ALFANUMS::           [ ALFANUM ]+ .
DEC_DIGIT::          0..9  .
OCT_DIGIT::          0..7  .
CHAR::               A..Z | _ | a..z .
--- end 2.2
```

2.6 Program-generation for the sub-formalisms

Our goal is to develop a program generator for the unifying formalism. In the schema below we indicate the complexity of recognition, parsing and transduction for the different sub-formalisms. We will start with the simplest formalism and will enrich it, step by step, with more evolved ones. In our classification we use a label for each sub-formalism, with a number of possible values. Because of its importance we will indicate separately 1. the handling of ambiguity and 2. forced sequencing of rules. Some combinations are not possible. We will then leave out a label, which is identical with <label>="no".

The simplest grammar-form which will be our starting point is formed by the start symbol in the lhs and a concatenation of one or more terminal symbols in the rhs (sometimes called a Chomsky type-4 grammar). This rhs then forms the language.

The labels are :
- Type (for Chomsky-type grammar). Possibilities:
- - type 4, 3, 2, 1 or 0
- Meta (for notation). Possibilities:
- - no, BNF, Regexpr, Automaton.
- Amb (for ambiguity). Possibilities during recognition:
- - no, back (for backtracking), par (for parallel parsing).
- Seq (yes for forced sequencing, no for otherwise)
- Arb (for one or more arbitrary endsymbols). Possibilities:
- - no, don't care, arb (the special case of one leading and one trailing arb we will indicate by "pattern"), line, range
- Var (for variables). Possibilities (combinations are possible) :
- - no, G (for global), L (for local), P (as parameter), A (for assignment from the text).
- Bool (for "Boolean" operators). Possibilities:
- - no, O (or), A (and), N (not).
- - combinations are OA, ON and OAN
- Tree (for treestructures in grammar and text). Possibilities:
- - no, implicit, explicit

With this classification we can reformulate our goal as the development of a program generator for the parameters Type=0, Meta=regexpr, Amb=par, Seq=no, Arb= don't care + arb + line + range, Var=LPA , Bool=OAN, Tree=explicit.

Type=4

This is the simplest form, as indicated above. The text is the same as the grammar. Recognition with the same complexity as Type=4, Arb=pattern.

Type=4, meta=regexpr

This is the recognition of regular expressions, consisting only of terminal symbols. Recognition in O(n) given in the textbooks, i.e. (Hopcroft and Ullman, 1979, pp.29-35).

Type=4, Arb=pattern, Seq=no

If the first and last symbol are arbs then this part of our schema concerns pattern matching. Without preprocessing recognition is done in time O(n*m), where m is the length of the keyword and m is the length of the input. Recognition strategies with pre-processing for pattern matching in O(n) or less are given by :
- (Knuth, Morris and Pratt, 1977) for one keyword in O(n); preprocessing creates tables
- (Boyer and Moore, 1977) for one keyword in less then O(n); preprocessing creates tables
- (Aho, Hopcroft and Ullman, 1974, ch. 9) and (Aho and Corasick, 1975) for a set of keywords in O(n) (as indicated: no implied sequencing); preprocessing implies construction of a FSA with output.

Type=4, Arb=pattern and don't care, Seq=no

Recognition by (Fischer and Paterson, 1974) in time $O(m.(\log n)^2.\log\log n)$ for one keyword.

Type=4, Tree=explicit, Seq=yes.

Here we are dealing with the recognition of tree-structures. Strategies are given by:
(Kron, 1975), (Overmars and Van Leeuwen, 1979), (Hardgrave, 1980), (Hoffmann and O'Donnell, 1982).

Type=2, Meta=BNF, Amb=no, Seq=no

These are the non-ambiguous context-free grammars.
There exists a large literature on subdivisions and on the recognition of this class of grammars. Construction in a systematic way of hand-coded recognizers may be done by the "recursive-descent" method.
Preprocessing may involve the automatic construction of automata, top-down (i.e. LL(k) grammars) or bottom-up (i.e. LR(k) grammars). The largest class of non-ambiguous cf-grammars which are automatically transformable into deterministic PDA's is formed by the LR(k) grammars. The literature may be found in the text-books. Bibliographies are contained in e.g. (Nijholt, 1983), (Burgess and Laurence, continually updated) and (Tokuda, 1981).
Because we will borrow a number of techniques of LR(k) parsing we mention here some important topics which play a role in the literature:

- the elimination of unit reductions ,
- the treatment of empty rules,
- generation of stack-instructions only if necessary,
- no transformations of the input-grammar,
- the handling of shift/reduce-reduce conflicts,
- the generation of shared code,
- the minimization of the number of generated states,
- error detection as soon as possible,
- some error correction,
- the number of necessary look-ahead symbols generated from the LR(0) machine (Kristensen and Madsen, 1981).

Type=2, Meta=Regexpr, Amb=no, Seq=no

These are the non-ambiguous cfg's with regular expressions, called "extended context-free grammars" (ecfg's). Madsen and Kristensen (1976) described transformations to transform ELR(k) grammars into normal LR(k) grammars. Heilbrunner (1979), Lalonde (1979 and 1981) and Purdom and Brown (1981) improved this method by directly generating PDA's from these grammars.

Type=2, Meta=BNF, Amb=par, Seq=no

These are the ambiguous cfg's. The literature on recognition is found in the textbooks. The most general method of Earley (1970) realizes an upper bound of time $O(n^3)$.
This upper-bound is theoretically improved by Valiant (1975), Graham and Harrison (1976) and Bouckaert, Pirotte and Snelling (1975) by making use of general methods for matrix-multiplication.
The method was rediscovered by Kay as "chart parsing" and is described extensively in (Winograd, 1983). Tomita (1986) describes a parser generator for fast execution with a dag-structured stack.

Type=2, Meta=BNF, Amb=par, Bool=ON, Var=A, Seq=no

Ken-Chih Liu (1981) describes an extension of cf-grammars with Boolean negations and variables which may get an assignment from the input-string.
He uses an extension of Earleys algorithm and suggests extensions of the LR-parsing technique in the deterministic case.

Type=2, Meta=Autom, Amb=back, Var=G, Seq=no

These are the ATN-grammars. Textbook: (Winograd,1983).
Most of the recognizers are written in LISP. Surrounding routines (e.g. semantic) must then also be written in LISP. The handling of ambiguity is generally done by back-tracking. Interpreters and compilers for ATN-grammars are described in (Bolc, 1983).

Type=2, Meta=BNF, Var=P, Seq=no

These are the attribute- and affix-grammars. Katayama (1984) described a recursive-descent method to translate attribute grammars into procedures. Recognition procedures may be found in (Courcelle and Franchi, 1982), Pohlmann (1983), (Jourdan, 1984), (Katayama, 1984), (Lorho, 1984), (Waite and Goos, 1984), (Meijer, 1986) and their references. Normally a sentence in the language described by the cfg is parsed. The nodes of the resulting parse tree are "decorated" with appropriate formulas expressing the attribute relationships, and then the attributes are evaluated. Problems arising here concern the possibility of circular evaluations.

Type=1 and 0, Meta=BNF, Amb=no, Seq=no

Deterministic type-1 grammars may be parsed by methods presented by Walters (1970) and deterministic type-1 grammars by Turnbull (1975). They make use of 2 stacks.

Type=0, Meta=BNF, Amb=back, Tree=implicit, Seq=yes

These are the transformational grammars. In general a transformational grammar consists of a cf-component, a general rewrite component and transformations on the parse tree. The construction of the parse tree is implicitly assumed. The transformations are usually ordered. A single program is written (Friedman, 1971) to generate sentences according to a transformational grammar. Some recognizing programs are known which attempt to parse by evaluating all possible continuations after each input symbol. The resulting complexity is of an exponential nature.

Type=0, Meta=BNF, Amb=back, Tree=explicit, Seq=yes

In order to overcome the computational difficulties of transformational grammars attempts have been made to write the transformations in an inverse way as tree-transduction rules with implied sequencing.
Chauche (1974) developed a system to perform these transductions in, it seems, polynomial time. It makes use of a combination of a regular and a context-free tree-transducer. Transduction occurs in 3 phases. The outcome of each phase has to be deterministic.

3. The rationale of the PTA

3.1 Introduction

In chapter 2 we gave an overview of the methods which are in use for the creation of programs for some of the sub-formalisms of the unifying formalism. The question is how to extend and to combine these methods in order to create a program generator for the whole formalism. In order to answer that question we will follow in this chapter an evolutionary trail through the different solutions which are given for the recognition and parsing of Chomsky type-2, -1 and -0 grammars. The natural continuation of this trail will lead us to the concept of the PTA, which will be defined formally in chapter 4. We will also explain the means by which the formal definition will be presented.

3.2 The evolution of automata and program generators for the grammars in the Chomsky-hierarchy

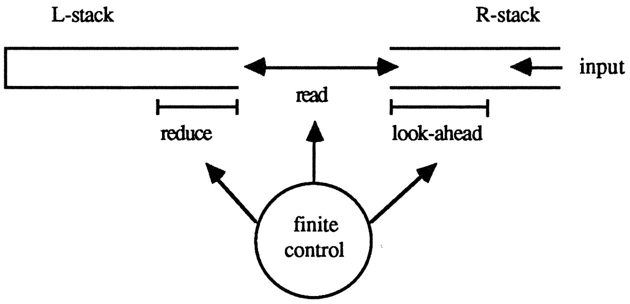
The usual goal for program generators for Chomsky type grammars is to generate a parser. They are therefore called "parser generators".
There is a close correspondence between the hierarchy of Chomsky grammars and the following formal automata:

sentences generated by a Chomsky grammar of type	can be recognized by a
4	FSA (Finite State Automaton)
3	FSA
2	PDA (Push Down Automaton)
1	LBA (Linear Bounded Automaton)
0	2SM (Two Stack Machine)

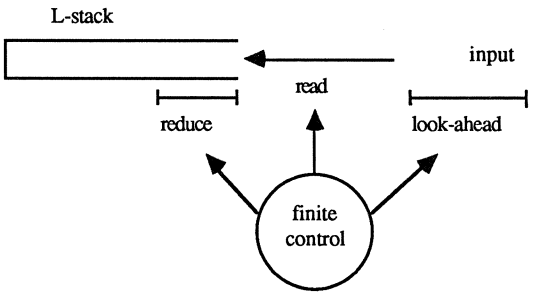
The trail which reaches a sub-class of type-0 grammars makes use of LR parser-generation techniques. This technique was initially developed by Knuth (1965) for a sub-class of the nonambiguous cf grammars. It can also be used for parser-generation for type-3 and -4 grammars (as we will show).
We will follow this LR-trail and will start with a short review of the methods of Knuth, Walters and Turnbull. They all worked with nonambiguous grammars. Then we will review the escapes that were made by Earley and Tomita in order to treat ambiguous cf grammars. Our own extensions will lead us to parser-generation for all type-0 grammars.
The discussion will be interspersed with the necessary definitions.

The machine models

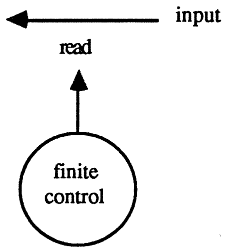
The formal machines which we listed above form a hierarchy, in the same way as the corresponding types of grammars. It will be slightly easier to start the explanation with a 2SM rather than with a FSA. The use of a 2SM for parsing can be depicted as follows (from Turnbull (1975)):



A 2SM is composed of an L-stack (a stack), an R-stack (really an output restricted deque (Knuth 1968)), and a finite control. Initially the input resides in the R-stack, beginning at the left end. The control is designed so that it either shifts a symbol from the left end of the R-stack and pushes it into the L-stack, or applies a rewriting rule to the top of the L-stack. The latter operation is known as a reduction. It entails removal of symbols from the top of the L-stack and insertion of some related symbols (related by the productions) into the left end of the R-stack.
Shift reduce parsers (Aho and Ullman, 1972) operate like this model. Because they are based on cfg's they may only push a single symbol into the R-stack during a reduction operation. Instead of using the R-stack the symbol can be pushed on the L-stack. In that case we are left with a PDA :



Various parsing techniques differ only in how the control operates and how it is constructed. Research into parsing based on formal grammars has been directed at increasing the power of the control and, consequently, enlarging the class of languages that can be parsed. LR(k) grammars correspond to the largest set of cf languages that can be deterministically parsed from left to right in this model, looking ahead a bounded number of symbols (=k) in the input. LR(k) grammars correspond to the class of cf languages recognizable by a deterministic PDA.
Still lower in the hierarchy are the type-3 and -4 grammars. For recognition they do not need a stack and we are left with :



In the parsing methods which we shall review the finite control consists of a number of states. In this case the last machine model consists of a FSA.

3.3 Informal introduction to LR(0) parser generation and parsing by an example

The essence of the LR-method is that we associate with a state in the finite control a set of grammar rules. Each rule is marked by a position (the "dot"). The combination of a rule with a position is called an item. The presence of an item indicates that a possible derivation may include the grammar rule of the item, based upon the information up to the dot. The compiler calculates possible sets of items ("itemsets"), which are associated afterwards with the states of the finite control.
The items in an itemset determine which symbols in the input may be expected. For each itemset possible actions on these symbols are calculated. An action can be a shift, a reduce, an accept or an error. The actions for an itemset are sampled into a so-called "LR-table". In this LR-table the individual items of the itemset are no longer present. In an LR table only one action may be specified for an input symbol. This condition accounts for a deterministic control. If more then one action is specified for a symbol in an itemset then that itemset is called inadequate. In that case an attempt can be made to resolve the inadequacy by the calculation of lookahead. If k symbols are sufficient to resolve the inadequacy for all itemsets then the grammar is called LR(k). If no such k can be found the grammar is called non-LR, and other approaches have to be followed.
The control is guided by the LR-table. This can be implemented by an interpreter. However, an LR(k) table can be translated into code for a PDA. In that case the control is that of a PDA.
In order to illustrate the calculation of the actions we present a simple example.

Suppose we want to construct an LR-table for the following cfg (for reference we number the rules):

```
1       S :: NP, VP.
2       NP :: Art , N.
3       VP :: V , NP.
4       VP :: V.
```

According to the convention which we adopted in chapter 2 the nonterminals are {S,NP,VP}, the start-symbol is S and the terminals are {V,N,Art}.
We augment the grammar by the rule

```
0       S' :: (, S, ).
```

The "(" and ")" are the reserved symbols for a tree. We introduce them as the natural begin- and endmarkers for the highest level in a, in principle, tree-structured input. We suppose that the start and the end of the input will be signaled by these symbols. The construction of the itemsets starts with the augmented rule with the dot in the first position. In order to avoid confusion we leave out the dot at the end of a rhs. For referencing we number the items as follows (superscripted numbers):

0 S' :: ^{1}(, ^{2}S, ^{3}) ^{4}.
1 S :: ^{5}NP, ^{6}VP ^{7}.
2 NP :: ^{8}Art , ^{9}N ^{10}.
3 VP :: ^{11}V , ^{12}NP ^{13}.
4 VP :: ^{14}V ^{15}.

<u>Itemset 1</u>
1: S" :: . (, S,)

Comment : in the first state only one terminal may be expected, the "(". There is only one item that shifts over that symbol. The resulting item has the dot moved over the symbol, signaling that an "S" may be expected. The resulting item forms the "core" of a new itemset, number 2. (The items which belong to the core are denoted above a dotted line.) We therefore create in the LR-table for itemset 1 and the symbol "(" the entry "shift 2".

(	:	shift	2

<u>Itemset 2</u>
2: S" :: (, .S,)

5: S :: . NP, VP
8: NP :: . Art, N

Comment : item 2 forms the core of itemset 2. An S may be expected now. Because the S is a nonterminal we add all the items with the S at the lhs and the dot at the leftmost position (we call this a "starting item"). It is item 5. In the items which are added (only 5) the dot may again stand for a nonterminal, here NP. Therefore we add all the starting items with NP at the lhs (in this case only item number 8). We repeat this process (adding items if they are not already present in the itemset) until no more starting items are added with the dot before a nonterminal. The set of all starting items which are added because of the presence of "S" is called the "closure" of S. (In general the closure of each nonterminal can be calculated before the process of generating itemsets starts.)
For each symbol after a dot we calculate the corresponding entry in the LR-table. On the symbols S, NP and Art we construct 3 new initial itemsets.We compare each of these initial itemsets with the core of already existing itemsets. If the test fails for an initial itemset it is placed into a queue of itemsets which have to be treated.
For itemset 2 we can now create the following entries in the LR-table :

S	:	shift	3
NP	:	shift	4
Art	:	shift	5

<u>Itemset 3</u>
3: S" :: (, S, .)

Comment : the dot stands before the endmarker. We generate on this symbol the entry "accept".

)	:	accept

Itemset 4

```
 6 : S :: NP, . VP
 --------------------
 11 : VP :: . V, NP
 14: VP :: . V
```

Comment : the core of itemset 4 is formed by item 6. Closure items are 11 and 14. On the symbol V item 14 will reduce and item 11 will shift. Because two actions are possible the itemset is called "inadequate". In this situation lookahead calculations may become useful.

V	:	shift 6
VP	:	reduce 1 (=rule 1)
V	:	reduce 4

The rule "VP :: V" contains one symbol at the rhs. We call this a "unit rule" and the reduction according to this rule a "unit reduction". We know beforehand that after reduction of this rule we will be back in this same itemset 4 and that we have to perform the reduction according to the rule "S :: NP, VP". It is therefore possible to combine the two reductions into one. In general there can be a chain of unit reductions within one itemset, ending with a reduce or shift for a rule with |rhs| > 1. In our case the LR table for itemset 4 would have three entries for symbol V :

V	:	shift 6
V	:	reduce 4,1

Itemset 5

```
 9: NP :: Art, . N
 ---------------------
```

N	:	reduce 2

Itemset 6

```
 12: VP :: V, . NP
 -------------------------
 8: NP :: . Art, N
```

Comment : a shift on Art generates an initial itemset with a core identical to that of itemset 5.

Art	:	shift 6
NP	:	reduce 3

The queue of initial itemsets is now empty.

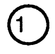
The resulting LR-table is (entries for symbols which are not mentioned are implicitly "halt"):

Statenr	symbol	shif t to state	reduce rule nr.	accept
1	(	2		
2	S NP Art	3 4 5		
3	)			x
4	V VP	6	4 1	
5	N		2	
6	Art NP	5	 3	

Example of parsing with the LR-table.

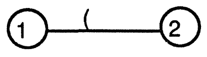
With the aid of the LR-table we will parse the sentence "(Art N V Art N)".
Because the table contains inadequate states we have to rely on some technique to resolve the inadequacies. One alternative is to calculate how many lookahead symbols are necessary. Another one is to use a parallel parsing technique. Because the PTA is based upon the last approach we continue with our example in that fashion.

We start in state 1.



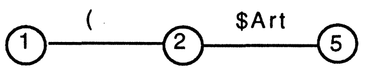
($Art $N $V $Art $N)

read (and shift



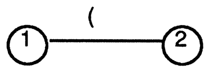
$Art $N $V $Art $N)

read $Art and shift



$N $V $Art $N)

read $N and reduce NP: $Art :$N



$V $Art $N)

shift on the lefthand symbol NP

$$\begin{tikzcd} \textcircled{1} \arrow[r, dash, "("] & \textcircled{2} \arrow[r, dash, "NP"] & \textcircled{4} \end{tikzcd}$$

$V $Art $N)

read $V and 1. shift
2. reduce S::NP VP
shift on the lefthand symbol S

$$\begin{tikzcd} \textcircled{1} \arrow[r, dash, "("] & \textcircled{2} \arrow[r, dash, "NP"] \arrow[dr, dash, "S"'] & \textcircled{4} \arrow[r, dash, "\$V"] & \textcircled{6} \\ & & \textcircled{3} & \end{tikzcd}$$

$Art $N)

read $Art and shift (in 6); (3) has no continuation and disappears

$$\begin{tikzcd} \textcircled{1} \arrow[r, dash, "("] & \textcircled{2} \arrow[r, dash, "NP"] & \textcircled{4} \arrow[r, dash, "\$V"] & \textcircled{6} \arrow[r, dash, "\$Art"] & \textcircled{5} \end{tikzcd}$$

$N)

read $N and reduce NP :: $Art $N (return in 6)
on the lefthand symbol NP reduce VP : $V NP (return in 4)
on the lefthand symbol VP reduce S :NP VP (return in 2)
on the lefthand symbol S shift

$$\begin{tikzcd} \textcircled{1} \arrow[r, dash, "("] & \textcircled{2} \arrow[r, dash, "S"] & \textcircled{3} \end{tikzcd}$$

)

read) and accept

3.4 Considerations about lookahead calculations

The construction of an LR(0) table is the basis for the construction of an LR(k) table. It is also the basis of the algorithm of Earley (1970) for general cf parsing in cubic time and of the construction of the automata of Walters and Turnbull, who generalized the LR(k) approach for deterministic type-1 and -0 grammars which we will review in the next subsections. It seems therefore appropriate to formalize the informal discussion in the previous example by a recapitulation of the basic algorithm, as it is given in (Aho and Ullman, 1977).
In this book we are concerned with the parsing of generalized, ambiguous type-0 grammars. We are therefore not interested in classes of grammars which can be parsed deterministically by using k symbols lookahead. In our PTA lookahead will be profitable only for optimization purposes.
In that respect it is useful to observe the relation between LR(0) and LR(k) table construction. In the original paper of Knuth (1965) each LR(k) item contains a lookahead string which is used in every algorithm related to the construction of an LR(k) table. In the subsequent literature on LR parsing this approach was more or less maintained. However, this is not necessary. The calculation of lookahead can be performed after the construction of an LR(0) table. The resulting simplification in the explanation of the construction of LR(k)-tables is worded by Heilbrunner (1981): "It is not a simple task to give a convincing explanation of the commonly used LR(k) definition. There are detailed tutorials on LR parsing which do not even state it (Aho and Johnson, 1974), (Houwink ten Cate 1974). The technical apparatus for a presentation of LR theory along the now traditional lines of (Aho and Ullman, 1972) involves a lot of tedious details which may prove the results but certainly obscures the ideas. A comprehensive and formal treatment using these methods is contained in (Geller and Harrison, 1977). We claim that basically simple ideas and a few clever tricks are sufficient to explain and prove correct in an intuitively clear and still formal way Knuth's

original LR algorithm (Knuth, 1965) and DeRemers's (1969,1971) and Pager's (1977) variants. We shall see that a grammar is an LR(k) grammar if and only if the straightforward nondeterministic bottom-up parsing algorithm for this grammar can be made deterministic by eliminating superfluous moves".
Superfluous moves are caused by inadequate states. Our LR treatment of ambiguous grammars in the Chomsky hierarchy will concentrate on the treatment of inadequate states in the generated LR(0) tables. Because the inadequacies can not all be resolved by lookahead calculations we will resolve them by efficient parallel parsing. Some lookahead calculations could be useful in order to improve efficiency, but they are not necessary.[1]
In the papers of Walters (1970) and Turnbull (1975) the original algorithm of Knuth (1965) for the construction of parsing tables is followed, with lookahead strings contained within items. Their algorithms for the construction of parsing tables collapse to the algorithm for the construction of LR(0) tables when lookahead calculations are not taken into account. Again, we are interested in the parsing of ambiguous type-0 grammars. It seems therefore appropriate to adopt the algorithm for the construction of LR(0) tables as our basic algorithm for the construction of the control of the PTA and to develop techniques for the calculation of lookahead for some of the inadequate states later on. In chapter 5 we will recapitulate the LR(0) parser-generation algorithm. We will not recapitulate the algorithms of Walters and Turnbull for the construction of LR(k) tables but will mention only the working of the control on the L- and R-stack that operates according to these tables.

3.5 The relation between the finite control and the LR(0) table according to Knuth, Walters and Turnbull

Knuth, Walters and Turnbull present the working of the finite control as a sequence of configurations.
A configuration of an LR processor is a pair whose first component is the stack contents and whose second component is the unexpended input. Walters extends this towards a configuration for a CS processor : a configuration has two stacks, L and R. The content of the L-stack is $S_0S_1...S_n$ ($S_0...S_n$ are states) and the content of the R-stack is $X\beta$. Turnbull adds the current state as a separate third component. The others denote the current state as the top of the L-stack, and we will follow that convention. X is the top symbol on R. For cf grammars $X\in T\cup\{)\}$ and $\beta \in T^*$), for cs- and type-0 grammars $X \in V\cup\{)\}$ and $\beta \in V^*$).
As for the LR case, the CS processor starts in the configuration (S_0,ω)), where ω in T^* is the input string. If the automaton reaches the configuration ($S_0S_1...S_n$,$X\beta$)) then it has reconstructed a rightmost derivation of ω from $x_1...x_nX\beta$, where $x_1...x_n$ in V^* is the reconstructed rightmost derivation up till now.

We will explain how the four instructions shift, reduce, accept and error in the LR-table direct the control for a CS processor. Again we do this in a manner which is adjusted to our examples and to our algorithms in the subsequent chapters.
A next move of the configuration is determined by the symbol at the top of the L-stack S_n (the current state), the symbol at the top of the R-stack X and the consultation of ACTION[S_n,X] in the LR-table. According to the result of the consultation the next move is

[1] With the extension to type-1 and type-0 grammars new ways for the treatment of lookahead become possible. For instance, Turnbull (1975) shows that in the extension to non-ambiguous type-1 and -0 grammars lookahead can be eliminated by adding more context to grammar rules. This can be done in an automatic way, based upon lookahead calculations for the LR(0) table.

a. shift S: $(S_0S_1...S_n,X\beta) \Rightarrow (S_0S_1...S_nS,\beta)$
b. reduce $Y_1..Y_m :: \gamma$: $(S_0S_1...S_n,X\beta) \Rightarrow (S_0S_1...S_{n-r},Y_1..Y_m\beta)$,
where $|\gamma| = r$. For cfg's m=1.

Remarks: 1. In the standard LR-algorithm for cfg's with lookahead calculations chains of reduces always lead to a shift; moreover, S_n will contain the reducing item with the dot at the end, which is the reason that the X will only be inspected as a part of the lookahead; in that case the next move is defined as

$(S_0S_1...S_n,X\beta) \Rightarrow (S_0S_1...S_{n-r}S,X\beta)$,
where ACTION[S_{n-r},Y_1] is shift S.

2. In the algorithms of Walters and Turnbull for type-1 and -0 grammars it is guarantied, by the calculation of lookahead, that ACTION[S,X] will be a shift.

c. accept : the control stops. S_n has to be an accepting state.
d. error : the control calls some error reporting-, and eventually, an error-recovery routine.

The algorithm for the control itself is very simple : it starts in the configuration (S_0, $a_1..a_n$)) and performs next moves as long as no accept or error shows up.

Walters proves the equivalence of CS(k) grammars and DLBA's : a set of strings is accepted by some DLBA iff it is generated by some CS(k) grammar. In his case the 2SM is reduced to a DLBA. He also proves that the sentences of a CS(k) grammar can be parsed in a time proportional to the length of their derivations.
The control which he constructs for CS(k) grammars need not halt for every input.
Turnbull (1975) studies type-0 grammars for which parsers can be constructed which will detect any errors as soon as possible. These parsers are guaranteed to halt. We quote from (Turnbull, 1975, p. 3-1) : "In general terms, our goal is to define and examine the class of languages that can be parsed deterministically in a single left to right scan without backtracking. Since backtracking and multiple passes are inherently inefficient operations, this class is the largest set of languages that are practical to use as programming languages."
Both Walters and Turnbull observe that the steps from one configuration to another can be thought of as grammatical productions, in reverse, operating on the configurations. If no instruction can be applied to a configuration, then an error has been found. Based upon this idea Walters reconstructs from a DLBA a CS(1) grammar, and proves the equivalence.

3.6 Earley parsing

We will now leave the deterministic trail and will give a short recapitulation of the algorithm of Earley for the parsing of arbitrary cfg's. It contains the same ingredients as the algorithm for the creation of LR-tables. The main difference is that itemsets are created in runtime in response to the input, and that, in LR terms, itemsets may be inadequate. In our description we will use some of the terminology of Tomita (1986). The original algorithm of Earley looks ahead k input symbols, just like the LR(k) algorithm. We fix k to be 0, for the same reasons as we summed up in the previous subsection on LR table construction.

An Earley-item is of the form

$<A::\alpha.\beta, f>$,

where $A :: \alpha\beta$ is a production, f is an integer, $0<=f<=n$.
Alternatively, we sometimes denote it as

$<p, j, f>$,

where p is the production A :: $\alpha\beta$, and j is an integer representing the position of the dot (j = $|\alpha|$). Note that an Earley-item has one more element, f, than an LR(0) item [A :: $\alpha.\beta$].

Earley's algorithm proceeds as follows (we assume that the input is $a_1..a_n$) :

```
- $I_0$ := <S' :: .(S), 0>
- for i := 0 to n do
  - for each item in $I_i$ do
    - if the item is of the form <A :: $\alpha.B\beta$, f> then do PREDICTOR.
    - if the item is of the form <A :: $\alpha.$, f> then do COMPLETER.
    - if the item is of the form <A :: $\alpha.d\beta$, f> then do scanner.
- if <S' :: (S)., 0> is in $I_n$ then accept else reject.
```

PREDICTOR (runs analogous to the LR procedure closure)

```
- for each production B :: $\gamma$ do
  - add <B :: .$\gamma$, i> to $I_i$.
```

COMPLETER (runs analogous to the LR procedure REDUCE)

```
- if a = $a_{i+1}$ then
  - for each item <B :: $\alpha.A\beta$, h> in $I_f$ do add <B :: $\alpha A.\beta$, h> to $I_i$.
```

scanner (runs analogous to the LR procedure SHIFT)

```
- if d = $a_{i+1}$ then add <A :: $\alpha d.\beta$, f> to $I_{i+1}$.
```

The runtime efficiency of Earley's algorithm is O(n3), but compared to LR-parsers (in the deterministic case) the constant factor is large, because every itemset is constructed in runtime. A second drawback of the algorithm is that the reconstruction of parses has to be done afterwards, while the parsing algorithm itself has the on-line property.

3.7 Tomita parsing

After completion of the development of our PTA we learned about the work of Tomita (1986), who describes an algorithm for the parsing of general cfg's with a "graph-structured stack", as a combination of LR(1) and Earley parsing.
The idea of this combination was already described by Lang (1974) for the case of recognition. Tomita describes also the maintenance of a "parse forest", as an efficient representation of a number of parse trees. We identified his algorithms as a part of our algorithm, treating the case "Type=2, Meta=BNF, Amb=par, Seq=no". At the end of his dissertation Tomita expresses the wish to extend his algorithm to be able to handle more formalisms.
His treatment of the stack and of parse trees during the parsing of ambiguous grammars went through the same evolutionary process as in our case.
LR parsing is totally deterministic. Therefore, a linear stack is sufficient and the parse tree (there can be only one) can be created on-line.
"Earley" parsing is deterministic, too. It handles all ambiguous parses in parallel. But parse trees are reconstructed only afterwards. Another drawback is the wasting of space for the on-line storage of all created itemsets and the administrative overhead. This drawback can be avoided by making use of a combination of LR and Earley parsing.
We will describe the evolution process which we indicated above by some quotations from Tomita.
LR table construction can be seen as a kind of preprocessing for Earley's algorithm. When a created "parsing table has multiple entries deterministic parsing is no longer possible and

some kind of non-determinism is necessary". "The simplest idea is to handle multiple entries non-deterministically. We adopt pseudo-parallelism (breath-first search), maintaining a list of stacks called a *Stack List*. The pseudo-parallelism works as follows. A number of processes are operated in parallel. Each process has a stack and behaves basically the same as in standard LR parsing. When a process encounters a multiple entry, the process is split into several processes (one for each entry), by replicating its stack. When a process encounters an error entry, the process is killed, by removing the stack from the stack list. All processes are synchronized: they shift a word at the same time so that they always look at the same word."
Earlier we employed this method in the same way in a parsing system called "LALRPAT" (briefly described in Van der Steen, 1981) that was in use a number of years in the computing department of the Faculty of Arts of the University of Amsterdam (Van der Steen, 1982, 1985).
"A disadvantage of the stack list method is that there are no interconnections between stacks (processes) and there is no way in which a process can utilize what other processes have done already. The number of stacks in the stack list grows exponentially as ambiguities are encountered...This can be avoided by using a tree-structured stack...Whenever two or more processes have a common state number on the top of their stacks, the top vertices are unified, and these stacks are represented as a tree, where the top vertex corresponds to the root of the tree. We call this a tree-structured stack. When the top vertex is popped, the tree-structured stack is split into the original number of stacks. In general, the system maintains a number of tree-structured stacks in parallel, so stacks are represented as a forest...So far, when a stack is split, a copy of the whole stack is made. However, we do not necessarily have to copy the whole stack: even after different parallel operations on the tree-structured stack, the bottom portion of the stack may remain the same. Only the necessary portion of the stack should be split. When a stack is split, the stack is thus represented as a tree, where the bottom of the stack corresponds to the root of the tree. With the stack combination technique described in the previous subsection, stacks are represented as a directed acyclic graph...It is easy to show that the algorithm with the graph-structured stack does not parse any part of an input sentence more than once in the same way. This is because if two processes had parsed a part of a sentence in the same way, they would have been in the same state, and they would have been combined as one process." "The graph-structured stack looks very similar to a chart in chart parsing. In fact, we can view our algorithm as an extended chart parsing algorithm which is guided by LR parsing tables."
About the efficient representation of a parse forest Tomita writes : "it is desirable for practical natural language parsers to produce all possible parses and store them somewhere for later disambiguation, in case an input sentence is ambiguous. The ambiguity (the number of parses) of a sentence grows exponentially as the length of a sentence grows. Thus, one might notice that, even with an efficient parsing algorithm such as the one we described, the parser would take exponential time because exponential time would be required merely to print out all parse trees (parse forest). We must therefore provide an efficient representation so that the size of the parse forest does not grow exponentially."
Then he describes two techniques for providing an efficient representation: sub-tree sharing and local ambiguity packing. Tomita claims that no existing system has adopted both techniques at the same time. With sub-tree sharing is meant : "If two or more trees have a common sub-tree, the sub-tree should be represented only once." This is performed by making comparisons in runtime.
With "local ambiguity packing" is meant : "two or more subtrees represent local ambiguity if they have common leaf nodes and their top nodes are labeled with the same nonterminal symbol. That is to say, a fragment of a sentence is locally ambiguous if the fragment can be reduced to a certain nonterminal symbol in two or more ways...The top nodes of subtrees

that represent local ambiguity are merged and treated by higher-level structures as if they were only one node." Again, this is handled by comparison in runtime.

So far Tomita. His ideas about the dag-structure of a stack and the sharing of parse trees were independently developed by us. "Sub-tree sharing" is performed automatically in our PTA. This will be shown in the next example of the working of a PTA where a highly ambiguous cfg with regular expressions is treated. The example also serves to demonstrate how the enrichment of the formalism with reports is treated, in combination with regular expressions. We do not perform "local ambiguity packing" because, for linguistic reasons, we want to keep intact the representation of all possible parses. Tomita motivated the "local ambiguity packing" for reasons of efficiency : in worst case situations the representation of parse trees would otherwise grow exponentially. In chapter 7 we will show for the PTA that the complexity of recognition is the same as the complexity of parsing, which is therefore not influenced by the building of parse trees.
With the aid of the PTA all sub-formalisms of chapter 2 can be treated, in combination with the compiler which we describe in the chapters 5 and 6. This extends by far the formalism of a cfg which Tomita is able to treat.

3.8 The extension of a 2SM to a PTA

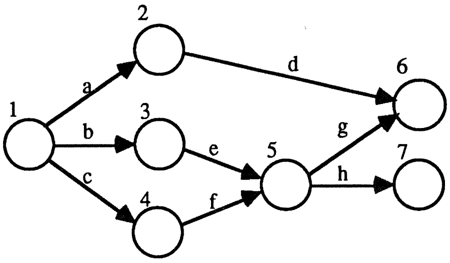
In order to be more precise we first give a number of relevant definitions and notations.

Definitions.
An unordered directed *graph* G is a pair (A,R), where A is a set of elements called *nodes* (or *vertices*) and R is a relation on A.
Unless stated otherwise, the term graph will mean directed graph.
A pair $(a,b) \in R$ is called an *edge* or *arc* of G. This edge is said to leave node a and enter node b. If (a,b) is an edge, we say that a is a predecessor of b and b is a *successor* of a.
A *labeling* of the graph (A,R) is a pair of functions f and g, where f, the node labeling, maps A to some set, and g, the edge labeling, maps R to some (possibly distinct) set. In many cases, only the nodes or only the edges are labeled. These situations correspond, respectively, to f or g having a single element as its range.
A sequence of nodes $(a_0,a_1,...,a_n)$, n>=1, is a *path* of length n from node a_0 to node a_n if there is an edge which leaves node a_{i-1} and enters node a_i for 1<=i<=n. If there is a path from node a_0 to node a_n, we say that a_n is *accessible* from a_0.
A *cycle* (or *circuit*) is a path $(a_0,a_1,...,a_n)$ in which $a_0=a_n$ (with n>=1).
The *in-degree* of a node a is the number of edges entering a and the *out-degree* of a is the number of edges leaving a.
A dag (short for directed acyclic graph) is a directed graph that has no cycles. A node having in-degree 0 will be called a *base node*. One having out-degree 0 is called a *leaf.*
Δ

Notations.
The nodes in a dag may be topologically sorted into a linear order (Aho&Ullman, 1972, p.45).
We will represent a dag by an ordered set of tuples (a,b), where a is a node and b is a set of tuples of nodes and symbols. If a dag D = $((a_1, ((b_{1,1}, s_{1,1}), (b_{1,2}, s_{1,2}), ..., (b_{1,n1}, s_{1,n1}))), ((a_2, ((b_{2,1}, s_{2,1}), (b_{2,2}, s_{2,2}), ..., (b_{2,n_2}, s_{2,n_2}))), ..., ((a_i, ((b_{i,1}, s_{i,1}), (b_{i,2}, s_{i,2}), ..., (b_{i,n_i}, s_{i,n_i}))), ..., ((a_m, ((b_{m,1}, s_{m,1}), (b_{m,2}, s_{m,2}), ..., (b_{m,n_m}, s_{m,n_m}))))$, then
1. $a_i <> a_j$ for $i <> j$

2. m is the number of nodes in D, $n_i <= m$ for all i
3. if in (a,b) the b is ε, then a is a leave node
4. $b_{i,j}$ is an a_k for some $k > i$
5. if a_i is no element of an (a,b) then a_i is a base node

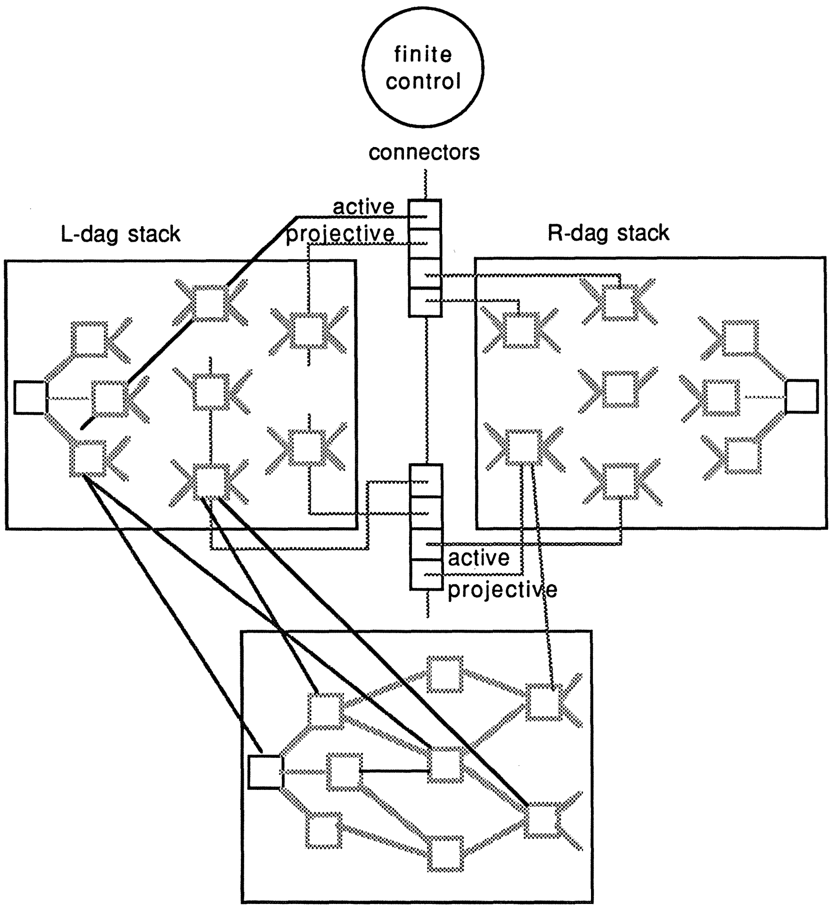
Example : the dag



is represented by D = ((1, ((2,a), (3,b), (4,c))), (2, ((6,d))), (3,((5,e))), (4, ((5,f))), (5, ((6,g),(7,h))), (6, ε), (7, ε));
1 is a base-node, 6 and 7 are leave-nodes.

The adjective "parallel" in "Parallel Transduction Automaton" has two meanings :
1. the execution of code which was produced automatically for a formalism in which, conceptually, each rule operates in parallel with other rules
2. the execution of this code on a parallel machine.
Meaning 1 has nothing to do with meaning 2. As we will see shortly the execution of the code can be done wholly on a sequential machine. But the machine can be organized in such a way that as much as possible can be done in parallel. The organization of the parallel processes does not stem from the inherent parallelism of the formalism but from the formal description of the instructions of the machine. In such a way an important gain in efficiency is achieved.

A PTA consists of 2 stacks in the form of a dag-structure (directed acyclic graph), a set of connectors and a parse forest.



parse forest

Each connector connects nodes in the L-stack with nodes in the R-stack. In principle a connector is a 4-tuple (active nodes$_L$, projective nodes$_L$, active nodes$_R$, projective nodes$_R$)[1]. (The indexes L and R refer to the L- and R-stack respectively). Active nodes project themselves, on the reading of an input symbol, to projective nodes. The number of possible projective nodes is limited and is calculated by the compiler. The operations on both stacks are in essence the same. The usual instructions for a stack, like pop, push, inspect top and change top are maintained. If an active node$_L$ is connected to an active node$_R$ then the intersection is taken of the two sets of possible continuation symbols. These symbols are found in the LR-tables for the compiled states which are attached to these nodes. For the symbols which are in common the corresponding actions in the LR-table are executed for the two nodes. After zero or more reduces a shift will follow, resulting in a push on the stack. This is performed by the creation of a new leaf in the dag and the creation of an edge from the last active node to that leaf node. All leaf nodes which result from actions taken for nodes in one active set (in the same dag) form a projective set. Before a new leaf node is created it is checked whether there already exists a leaf node in the projective set with the same number

[1] We will see later that connectors also have references for variables and the lexicon.

of a compiled state. In that case no new leaf node is created but an edge is formed from the last active node (the last one in a possible chain of reduces) to that already existing leaf node in the set of projective nodes. (This corresponds with "Earley parsing".) A reference to the resulting set is stored in a new connector.

A third dag is automatically maintained (if necessary) : the parse forest. This parse forest will be built on-line and reflects the current status of all valid parse trees. In the case of TDG's (transduction grammars) it will be used also as an association list for the nonterminals which appear in a lhs. Evaluation of these nonterminals will only happen when necessary ("lazy evaluation").
In the parse forest the report- and build-output find their natural, but temporarily, place. If the parse forest has become one single parse tree then this output will be brought automatically to the outside world and released from the parse tree.
The way of storing of the report- and build-output may be called : "deep binding". It is not necessary to reference these values until they have to be brought to the outside world. This in contrast with the variables. They are stored with "shallow binding", with a copying of the reference to their values from stacknode to stacknode.
Garbage collection is done entirely "on the fly" by keeping reference counts for each dagnode and its associated substructures. When the PTA has treated the endmarker in the input no datastructures are left.
The reference counts play a special role during transduction : when the reference count of a stacknode in L becomes 0 this is an indication that at that point no more transductions can be performed : the associated symbol will then be brought to the output file, together with the coordinates of its neighbours. In the case of ambiguous TDG's a dag with symbols from translations is in that manner brought to the outside world.
The translation is then produced with a finite delay (see also section 2.2.1). With finite delay we mean that output of the transducer becomes available after the reading of a finite number of input symbols, while the total number of characters in the input can be indefinite.

It is now possible to relate to the PTA the machine models which we described above and the parsing methods which make use of them.
The PTA can be seen as a general machine model which can be reduced to the former machine models. The 2SM consists of linear L- and R-stacks and of the finite control. A LBA, a PDA and a FSA are simplified models of a 2SM. The PTA extends the 2SM in the following ways :
- the L- and R-stack both become a dag
- a third dag is added for the storage of a parse forest
- connectors are added which connect sets of active and projective nodes in the L-and R-dag with each other
- storage is added for reports, builds and variables.
The purpose of the sets of active and projective nodes is that they serve to maintain a polynomial runtime complexity for each connector. If there is only one stack (the L-stack) only one connector is functioning. In that case we get back Tomita's parsing method. The dag-structure was chosen because of its efficiency in processing. The runtime behaviour for cf recognition with one stack is O(n^3). General rewriting makes use of both stacks.
The Parspat compiler only generates code for the smallest machine model that is possible. For the runtime behaviour the lowest bounds are valid which are achieved at this moment in theoretical research for the different sub-formalisms (as far as constructive methods are concerned). Furthermore, the PTA can be influenced by user-heuristics in order to decrease the number of ambiguous transductions.

These heuristics concern the treatment of inadequate states. In the LR-table of an inadequate state there may be denoted for a symbol one shift and a multiple number of reduces, to be subdivided in reduces for cf-rules and type-1 or -0 rules. The user of the PTA may indicate with global switches if a choice has to be made, and which kind of choice.

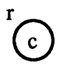
Code will be generated for the following (possibly combination) of sub-formalisms
- for a FSA: Chomsky-type= 3 or 4 with regular expressions, Arb= don't care and arb and line, Trees, Booleans;
- for a PDA : Chomsky-type= 2 with regular expressions, no ambiguity, don't care and arb and line, Trees;
- for a PTA with 1 dag-stack: Chomsky-type=2 with regular expressions, all ambiguities, don't care and arb and line, Trees, Booleans, variables;
- for a PTA with 2 dag-stacks: Chomsky-type=0 or 1 or transduction with regular expressions, all ambiguities, don't care and arb and line, Trees, Booleans, variables.

In the next subsection we will illustrate in an informal way the working of the PTA by a few examples. In these examples we will depict the different parts of the dags. In order to make the figures understandable the following explanations are given.

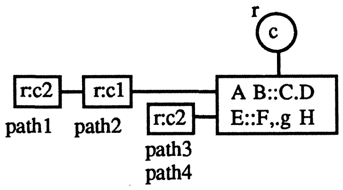
First we introduce for a dag the concept of a condensed notation. The notation of a path through such a condensed dag is called a condensed path.
In a condensed notation we leave out the symbols and denote for a node zero or one successor. For instance, the dag in the example above can be represented as
CD= ((1,2), (2,6), (3,5), (4,5), (5,6), (6,ε), (7,ε)).
The base-nodes are now 1, 3, 4 and 7 and the leave-nodes are 6 and 7.
A condensed path contains a pair with the begin- and end-node and it contains zero or more pairs of exceptions : if the successor of a node in the path is different from the successor which is named in the condensed notation of the dag then the node and its successor in the path have to be denoted as an exception.
Thus the path (1,3,5,6) is represented as a condensed path by ((1,6),((1,3))). The begin and the end of the condensed path are 1 and 6; at 1 has to be chosen as its successor 3 and not 2, which was specified in the condensed notation of the dag.
The use of a condensed notation is motivated by reasons of economy: long paths in a dag with nodes with low degrees are referenced substantially shorter.

In the PTA the L- and R-stack and the parse forest have the structure of a dag. The nodes of the dags contain information, which will be described precisely in the next chapter. We now give a short overview together with a pictorial representation in order to understand the examples.

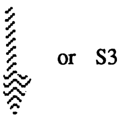
For the dags of the L- and R-stack we use the following pictures.



a node with label r ; it contains the number c of a compiled state; the numbers c and r are positive for the L-stack and negative for the R-stack; sometimes we call a node in the L- or R-stack a "runstate"

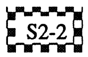


a node contains the numbers of the items of the compiled state (but not the starting items); in order to be clear we write out the text of an item; connected to an item are references (c1 and c2) to the runstates which have to become visible ("active") after a reduce (or "return") of that item; if necessary there are attached to each reference to a returnstate one or more descriptions of paths through the parse-forest

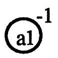


indicates a description of a path in the condensed form; if the path consists of a single node in the parse forest then the label of that node ("S3") is denoted; arrows are "coloured" by some pattern which corresponds to the pattern of a path which is depicted in the parse forest.

For the dag of the parse forest we use the following pictures.



A node in the parse forest which contains a nonterminal. In the dag of the parse forest the edges are not labelled, but only the nodes. The label consists of the nonterminal concatenated with the position of the input pointer. If there are more nodes with this label a negative number is attached to the node which, together with the label, makes the labeling unique. The rim of the rectangle has the same pattern as the path which spells out the parse tree in the parse forest associated with the nonterminal.

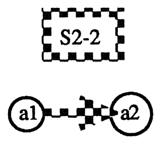


A node in the parse forest which contains a terminal. The same conventions hold as for a nonterminal. The negative number is, for reasons of space, written outside.

A node in a parse forest may contain information. In one example the information about reports is shown in this way

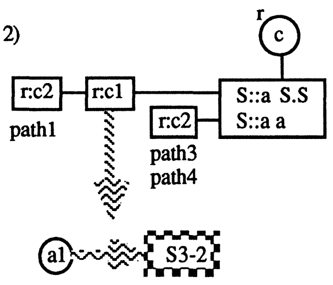
The parse forest is internally represented in a condensed notation. We do not show the edges, which are more or less chosen arbitrarily (depending on the input). Paths are indicated by arrows, pictured with some pattern. For instance :

1)



to the nonterminal S, recognised after the reading of the 2nd character in the input, belongs a (sub)tree consisting of the nodes a1 and a2.

2)



in an L-stack a partial parse tree, belonging to item S::a S.S , is indicated by an arrow.
After a reduction a return will be made to nodes c1 and c2.

3.9 Example of the working of a PTA for ambiguous cfg's with regular expressions and reports

This example will demonstrate the working of the instructions on the dag-structured L-stack, the on-line construction of the parse forest and the "deep binding" of report-output.
It concerns parsing according to the grammar :

```
0:      S' :: S {R:4} , ) .
1:      S :: a , [S {R:1}]* {R:2} , a {R:3} .
```

(Remark : in order to save space we leave out the beginmarker "(".)
The compiled LR(0) table for this grammar is, per state (the superscript digit marks an item) :

<u>State 1 :</u>
items: S' :: ^{1} S {R:4} ,) .
S :: ^{4} a , [S {R:1}]* {R:2} , a {R:3} .
table-entries:

a	:	shift 2
		report 2 for the itempair (from-item, shifted-item) = (^{4},^{6})
S	:	shift 3
		report 4 for the itempair (^{1},^{2})

<u>State 2 :</u>
items: S :: ^{4} a , [S {R:1}]* {R:2} , a {R:3} .
S :: a , ^{5} [S {R:1}]* {R:2} , a {R:3} .
S :: a , [S {R:1}]* {R:2} , ^{6} a {R:3} .
table-entries :

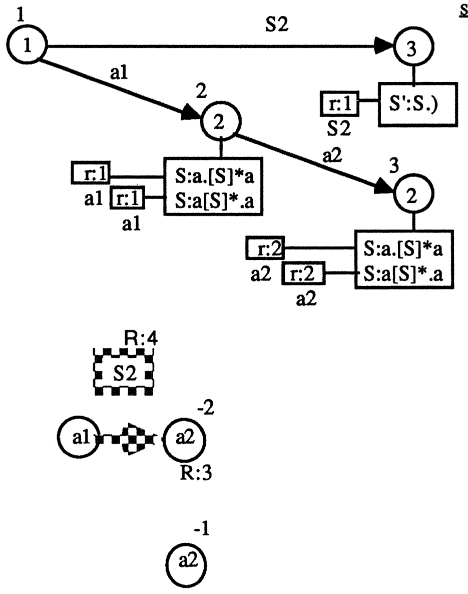
a	:	shift 2
		report 2 for the itempair (^{4},^{6})
S	:	shift 2
		report 1 for the itempair (^{5},^{5})
		report 1 and 2 for the itempair (^{5},^{6})
a	:	reduce rule 1 (in runtime : "go back to the node on the stack where this item originated and perform a shift on a S")
		report 3 for the itempair (^{6},^{7})

<u>State 3 :</u>
items: S' :: S {R:4} , ^{3}) .
table-entry :

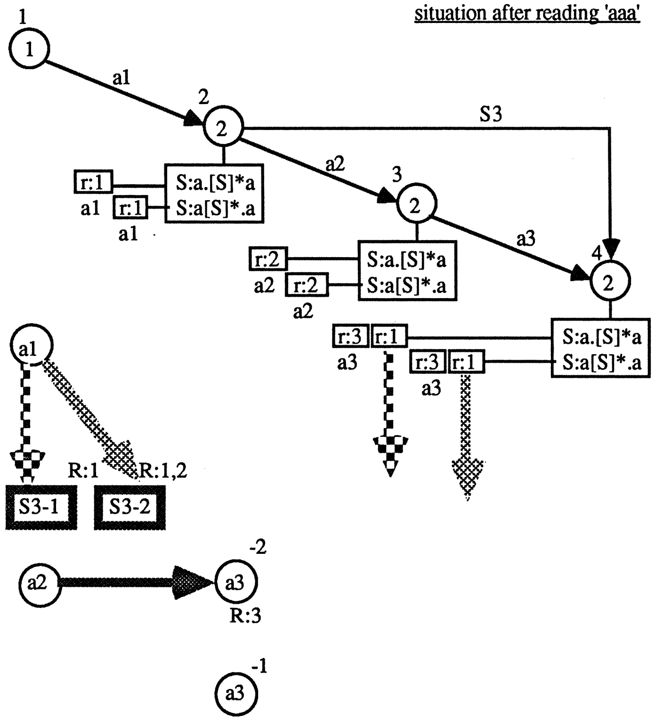
)	:	accept

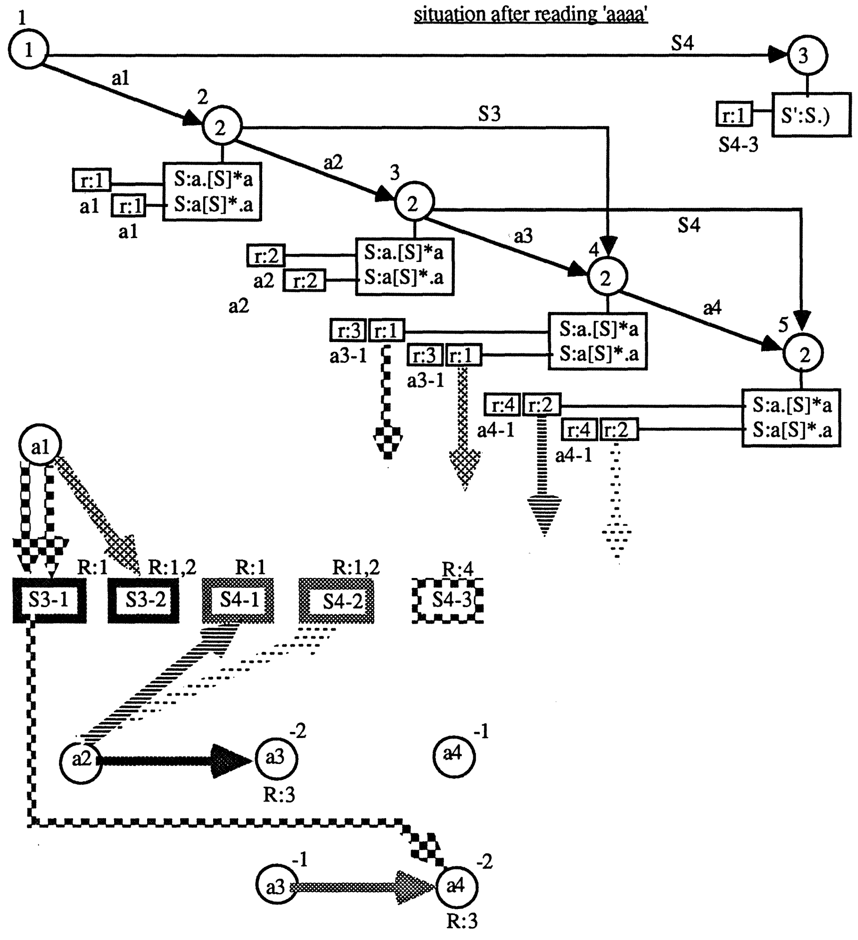
We assume that the input is "aaaaaa)". The L-stack is depicted in 6 situations, after reading "aa" until "aaaaaa)".

situation after reading 'aa'

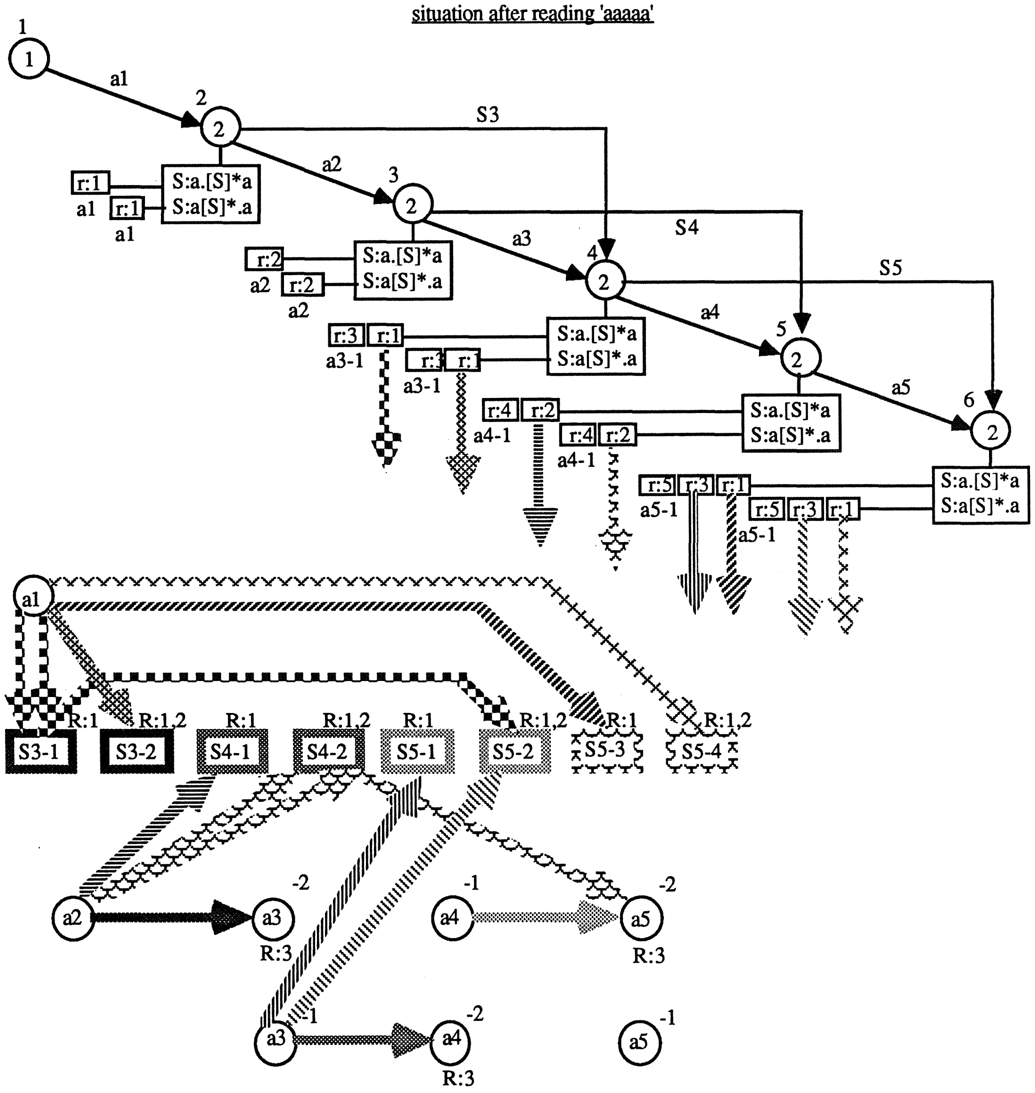


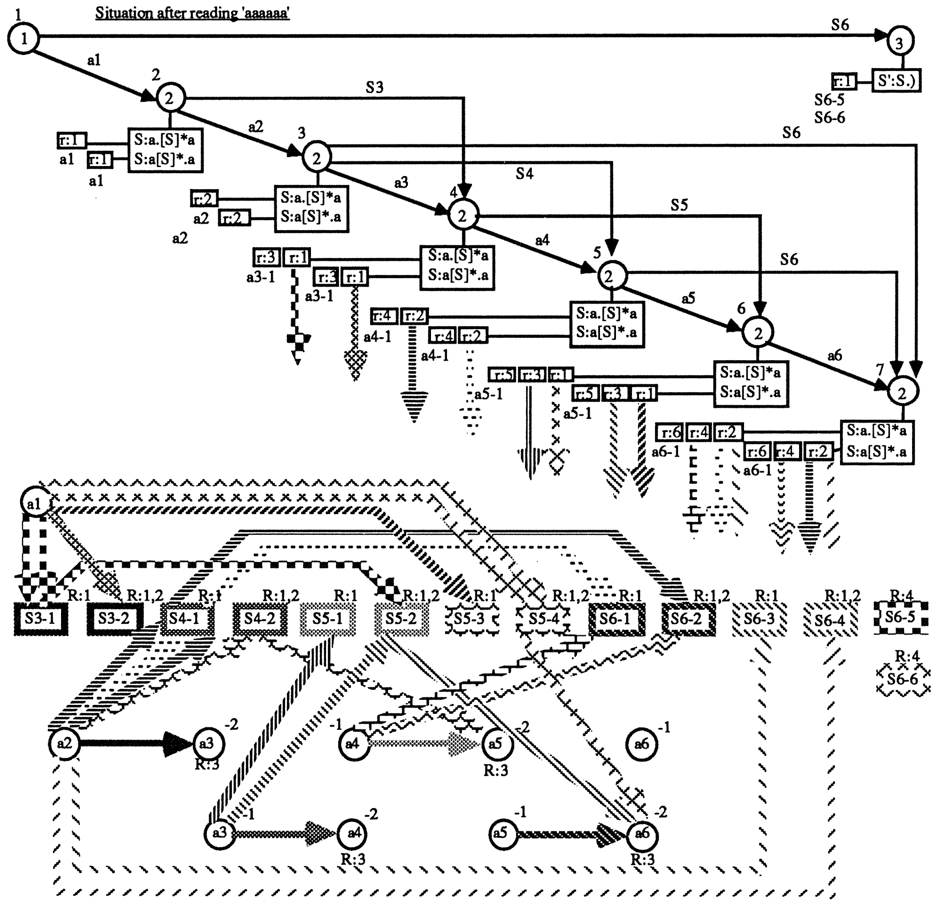
situation after reading 'aaa'



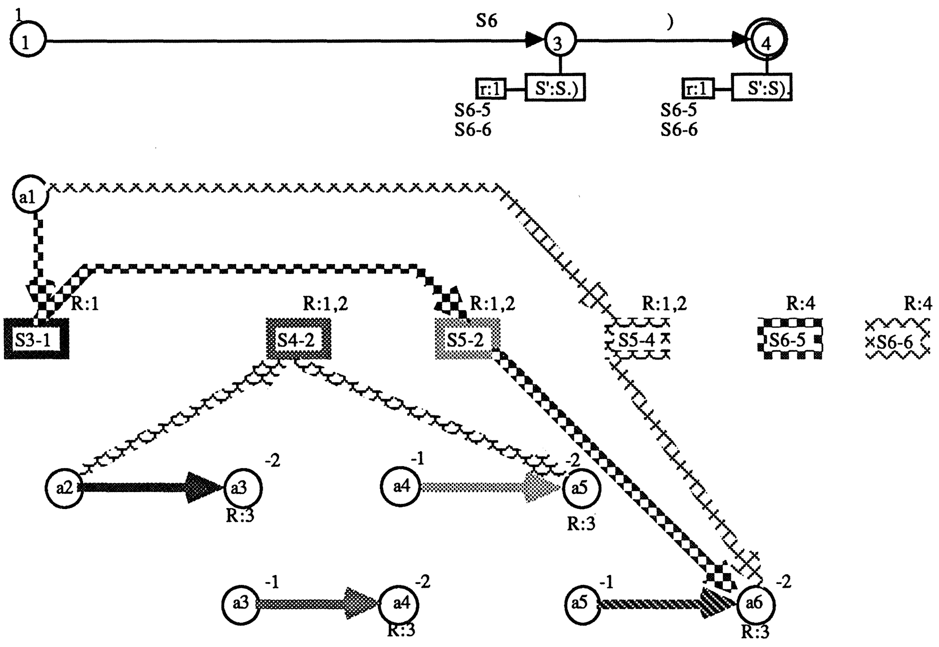

situation after reading 'aaaa'
1
S4
S3
S4
a1
a2
a3
a4
2
3
4
5
r:1
S':S.)
S4-3
S:a.[S]*a
S:a[S]*.a
r:1
r:2
r:3
r:4
a1
a2
a3-1
a4-1
R:1
R:1,2
R:4
S3-1
S3-2
S4-1
S4-2
S4-3
-1
-2
R:3

situation after reading 'aaaaa'

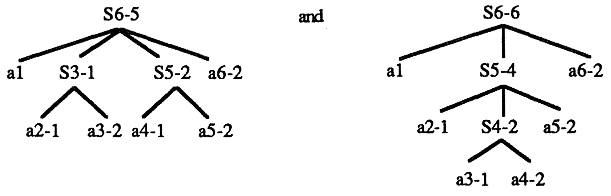



Situation after reading 'aaaaaa'
1
2
3
4
5
6
7
a1
a2
a3
a4
a5
a6
S3
S4
S5
S6
S:a.[S]*a
S:a[S]*.a
S':S.)
r:1
r:2
r:3
r:4
r:5
r:6
a3-1
a4-1
a5-1
a6-1
S6-5
S6-6
S3-1
S3-2
S4-1
S4-2
S5-1
S5-2
S5-3
S5-4
S6-1
S6-2
S6-3
S6-4
R:1
R:1,2
R:3
R:4
-1
-2

Situation after reading 'aaaaaa)'



And thus are represented the two parses



which are communicated as S (a S (a a) S (a a) a) and S (a S (a S (a a) a) a) .
The corresponding reports are communicated in postfix sequence as 3,1,2,3,1,2,3 and 3,1,3,1,2,3 .

3.10 Transduction with a PTA; example

We will now discuss informally how transductions are performed with the PTA.
On-line recognition with a 2SM can be extended towards transduction with finite delay. We compute LR-tables for the lhs's and the rhs's of the rules. We choose an example for which the generated states are adequate. In that case the dag-structured L- and R-stacks of the PTA remain flat. For the case of ambiguous type-0 grammars the reader may imagine, for the 2 stacks, the dag-structures as they were shown in the former example.

Our grammar is :

b , a :: a , b .

The compiled LR(0) table for the *left hand sides* of the grammar is :

<u>State -1 :</u>
items: -1 b , a :: a , b .
table entry for shifts: generate "b" and goto state -2 (in runtime : "push state -2 on the R-stack")

<u>State -2 :</u>
items : b , -2 a :: a , b .
table entry for reduces: generate "a" and reduce (in runtime : "go back to the state where this item originated")

The compiled LR(0) table for the *right hand sides* of the grammar is, per state :

<u>State 1 :</u>
items : b , a :: 1 a , b .
table entries :

a	:	shift 2 (in runtime : "push state 2 on the L-stack")
. (every other character)	:	shift 1

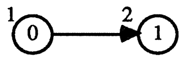
<u>State 2 :</u>
items : b , a :: 1 a , b . (all transduction rules may start in each state)
b , a :: a , 2 b .
table entries :

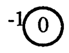
a	:	shift 2
b	:	reduce and put state -1 on the R-stack
.	:	shift 1

The entries on a stack have a reference count. If a reference count becomes 0 the entry is deallocated. If an entry at the bottom of the L-stack is deallocated, then its associated symbol is emitted. By assigning unique labels to the emitted symbols (e.g. increasing integers) a dag may be outputted (in the case of an ambiguous grammar).
In our example only the essential entries are shown. Not shown are the items in each entry which reference the entry where they originated, like in the previous example. If there are no more items which reference an entry than that entry will be deallocated.

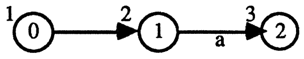
We show the on-line operation of the generated parser for the input "aab.."

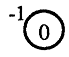
<u>L-stack :</u> <u>R-stack :</u>



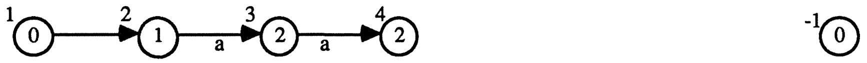


Read 1st symbol 'a' : shift to state 3(2)

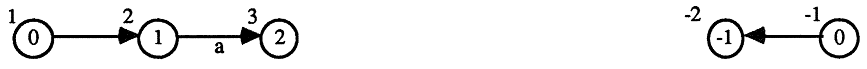




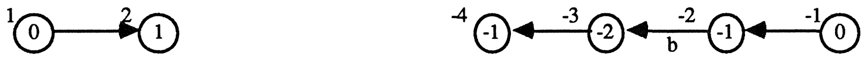
Read 2nd symbol 'a' : shift to state 4(2)



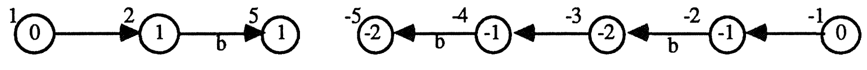
Read 3rd symbol 'b' : reduce and put state -2(-1) on the R-stack



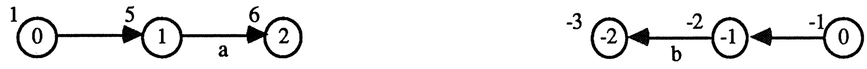
Generate from state -2(-1) symbol 'b' as input for state 3(2)
This causes a shift in state -2(-1) to state -3(-2) and a reduce in state 3(2)
Therefore put state -4(-1) on the R-stack



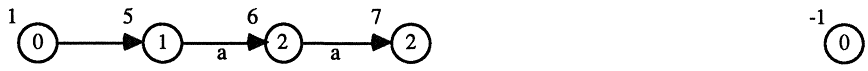
Generate from state -4(-1) symbol 'b' as input for state 2(1)
This causes a shift in state -4(-1) to state -5(-2) and a shift from state 2(1) to state 5(1)



State 2(1) is not longer referenced. The symbol 'b' will be emitted.
Generate from state -5(-2) symbol 'a' as input for state 5(1)
This causes a reduce in state -5(-2) and a shift from state 5(1) to state 6(2)



Generate from state -3(-2) symbol 'a' as input for state 6(2)
This causes a reduce in state -3(-2) and a shift from state 6(2) to state 7(2)



The 2nd stack is empty. We resume reading from the input.

Up till now the emitted output is :

1 b 5

where the superscripted digits denote the unique labels of the entries which were deallocated. Two a's are waiting on the stack for a possible transduction.

Note that the entries on the R-stack act as generators of symbols. That is the reason why regular expressions, arbs, lines, ranges and Boolean negations at the lefthand side of type-0 rules cause no problems[1].

[1] However, up till now we did not meet an application for them.

4. The definition of a Parallel Transduction Automaton

In this chapter we will formally define the PTA. First we will explain the means by which the definition will be given. The definition itself starts with an enumeration of the essential parts of the PTA. These parts may be grouped within three main categories. The first category concerns the structure of the elements of a configuration. The second category describes the processing within the PTA. In the third category output with finite delay and general pruning ("on the fly garbage collection") will be defined.
After that we will discuss a number of related topics: the use of the PTA for the generation of sentences, the problem of undecidability and the suitability of the PTA for implementation on parallel hardware.

4.1 Introduction to the formal description of the PTA

As our formalism for definition we choose the unifying formalism of chapter 2.
The working of the PTA will be described by transitions of configurations. With the help of cf grammar rules the structure of elements of configurations can be described concisely. With the notations for patterns and trees and with the possibility of general rewriting it provides for a short, but precise formulation of the working of instructions and of the process of pruning. A rigorous proof of the correct working of the PTA falls outside the scope of this study. However, with the non-procedural description of the rewriting of configurations such a proof will become less complicated.
For each possible instruction we will describe the corresponding transition, in the same way as Walters and Turnbull did. However, with the dag-structure for the two stacks and the parse forest this is more complicated. In order to be formal and at the same time understandable the description will consist of four components:
- an informal description
- a graphic representation
- a formal, non-procedural, rewriting of configurations; for the rewriting we will use a slight extension of the unifying formalism
- the procedural description which was programmed in the system Parspat, here given in pseudo-code.

The names of symbols will be uniform for all four components.
The definitions will be interspersed by comments within comment-brackets.

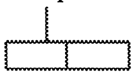
About the informal description

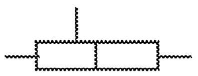
The informal description serves as a functional description of the purpose of the instruction and its relation to other instructions. We will abbreviate
- "a reduction of a rule with $|\text{lhs}| = 1$" by "a cf reduction",
- "a reduction of a rule with $|\text{lhs}| > 1$" by "a cs reduction".

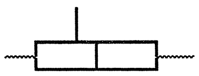
About the graphical description

We will accompany a non-procedural description by a pictorial representation in a one-to-one correspondence [1]. This correspondence is obtained by the following conventions.
- Correspondence with regular expressions :
  - in general a dotted line means : possibly present
  - if a member of a list is not necessarily the only one this is indicated by dotted fragments of lines to its left and right
  - if a member of a list is definitely not the only one the fragments of lines are solid
- Correspondence with don't cares and arbs :
  - if the value of a field of a record is not referenced (in the picture) it is not mentioned
  - only the elements which are necessary are drawn
- The 'next' of a list-element is, in general, not shown as a separate field
- A nil-pointer is indicated by a "/" .

Examples :


maybe one record


maybe a list of records


a list of 1 or more records

The correspondence with the informal description can be derived from comments in the picture. They are placed between curly brackets. In that way values may be denoted which are of no use for the formal description but which may clarify the notion of the reader.

About the non-procedural description

The rewriting of configurations will be formally described with the aid of the unifying formalism. It provides for a concise, but exact, description of the changes in a configuration.[2] The applicability of a rewriting depends solely on the matching of the rhs of a rule. For the purpose of the rewriting of configurations we extend the unifying formalism with the "reference" which has a direct correspondence with the "pointer" in Pascal: "^T1" means : a reference to the notion T1.
To be short, we will abbreviate

$[(\alpha)]^*$ by $(\alpha)^*$ and $[(\alpha)]^*$ by $(\alpha)^*$.

$\mathbb{N}$ means "integer". When the name of a variable V is prefixed by the name of a variable W which has been "declared" by a cf rewriting rule, then V has the same type as W and needs no further declaration.

[1] These pictures have been created by a picture-editor (MacDraw). In principle a formal description can be derived from the pictures. This scheme of thought suggests the realization of automatic programming by drawing pictures. Work along this line of thought is done also by e.g. Lewerentz and Nagl (1984). They suggest the use of "programmed sequential graph grammars". Here, sequential means that rewriting steps are applied one after the other. In our approach no sequencing is needed, provided that enough context is given for disambiguation.

[2] One more reason was to explore the possibility of writing programs solely in the formalism of general rewriting rules and to discover which transduction formalism and shorthands are desirable in that respect. We hope to be able in the future to compile all the rules together and to produce in that way an implementation of the PTA with the same behaviour as the one that is implemented according to the procedural descriptions. At this moment this is inhibited by some sequential processing that is assumed within the processors PL and PR.

The notation A:(α)* in a new configuration, where A is a nonterminal, means that the references within α have to be taken within the scope of the rewriting of an A in the old configuration; this notation effectively denotes the rewriting of a list.

About the procedural description

In the procedural description we assume sequential processing. This will have some consequences for the correspondence with the non-procedural description.
We prefer to write algorithms as short as possible. The following conventions are adopted :
- "begin"'s and "end"'s are not explicitly denoted but are implied by indentations
- each statement is preceded by a "-"
- comments are written between "{" and "}"
- if "st_i" denotes a statement then the following list shows our notation for the usual constructs:

```
(a) (type) procedure P(p1, p2, ..., pi) b) - if expr1      (c) - case var1 of
    - stn                                   - stn                - a1:    - stn
    ...                                     ...                           ...
    - stm                                   - stm                         - stm
    {end of P}                            - else                 - a2:    - stk
                                            - stk                         ...
                                            ...                           - stl
                                            - stl

(d) - for i := 1 to n do              (e) - for all a in A do  (f) - while expr1 do
      - stn                                 - stn                    - stn
      ...                                   ...                      ...
      - stm                                 - stm                    - stm
```

4.2 Formal definition of a PTA

A PTA consists of a 11-tuple PTA = (L, L_0, R, F, C, PT, PC, PL, PR, PF, PP) where :

- L is a dag with nodes of the type NODE; the type NODE is described hereafter by a description grammar ; an alias for NODE is : RUNSTATE; L_0 is the base node of L
- R is a dag with nodes of the type NODE
- F is a dag which forms the parse forest
- C is a finite set of connectors between L and R; the type connector is described hereafter by a description grammar
- PT is a processor for the PTA which activates PC
- PC is a processor for connectors in C which activates PL and PR
- PL is a processor which activates instructions in QL (to be defined hereafter)
- PR is a processor which activates instructions in QR (to be defined hereafter) and which activates also PF
- PF is a processor which activates instructions which operate on F
- PP is a processor for the pruning of datastructures.

A PTA is driven by a 8-tuple P = (QL, QL_0 , QR , QR_0, I , NI, T, ZCR) where :

- QL is a finite set of states QL_0, QL_1, ..., QL_n; QL_0 is the initial state; with each state is associated a scanner-table; with each entry in the scanner-table is associated a program with instructions from I; the structure of a program is described by a grammar
- QR is a finite set of states QR_0, QR_1, ..., QR_m; QR_0 as the initial state; QL and QR are disjunct; with each state is associated a scanner-table; with each entry in the scanner-table is associated a program with instructions from I; the structure of a program is described by a grammar
- I is a set of instructions, used in the programs; they are written as rewritings of configurations of a PTA; we will define the instructions which are necessary for the interpretation of a U-grammar
- NI is a finite set of nonterminal symbols and intermediate symbols
- T is a set of terminal symbols (not necessarily finite)
- ZCR is a set of variable, cooperation and report symbols.

The elements of P may be compiled from a U-grammar. Symbols are represented as integers. The representation of symbols in NI and ZCR may not overlap with the integer representation of symbols in T. This has to be guaranteed by the compiler for a U-grammar.

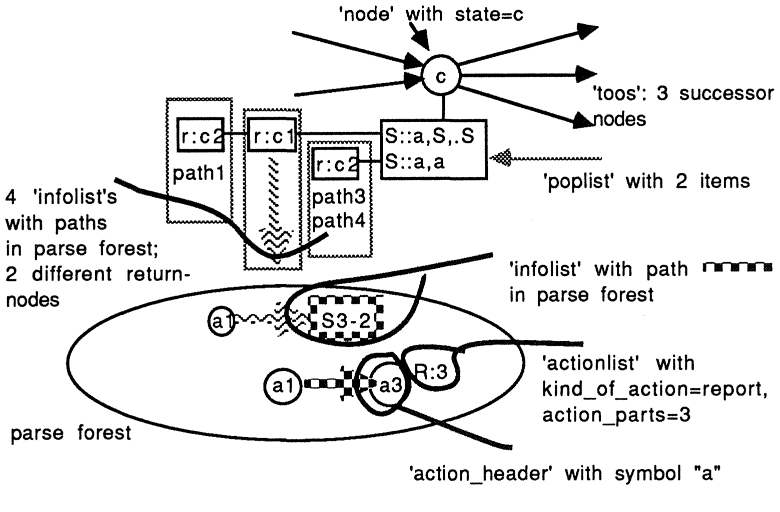
4.2.1 The structure of elements of configurations

A configuration of the PTA will consist of the dags L and R, the connectors C and some pointers within these elements. In the following subsections we will define the structure of L, R and C. L and R will contain references to the parse forest F; F is therefore implicitly part of a configuration. We will discuss the structure of F separately.

4.2.1.1 Nodes of L and R

Informal description of a node

In the following figure we repeat from chapter 3 a part of the pictorial representation of a piece of a node with references to the parse forest. We add to it the name of the notions by which we will describe the different parts in the description grammar for a node.


'node' with state=c
c
'toos': 3 successor nodes
S::a,S,.S
S::a,a
'poplist' with 2 items
r:c2
r:c1
r:c2
path1
path3
path4
4 'infolist's with paths in parse forest; 2 different return-nodes
'infolist' with path in parse forest
a1
S3-2
a1
a3
R:3
'actionlist' with kind_of_action=report, action_parts=3
parse forest
'action_header' with symbol "a"

Formal description of a node

L :: (L_0, NODE_L*).
R :: (R_0, NODE_R*).
L_0 | R_0 | NODE_L | NODE_R :: NODE.

NODE :: (refcount, ^TOOS, statenr, output-number, ^POPLIST, ^ASSOC) .
        refcount | statenr | output-number :: $\mathbb{N}$.
{ in "toos" a list is maintained to all successors of this node, together with the symbol on which the shift to the successor occurred; the symbol is, in case of a nonterminal, accompanied by its parse(s), stored in infolist_symbol }

TOOS :: (^NODE, symbol, ^infolist_symbol) * .
        symbol :: $\mathbb{N}$.

POPLIST :: (refcount, item, ^infolist) * .
        item :: $\mathbb{N}$.
{ in "poplist" all shifted items find their place, together with the partial parse(s) which they, up till now, covered }

INFOLIST :: (refcount, returnstate, nonterminal, PATH, ^VARIABLE_LIST,
                LEX_INFO)* .
        returnstate :: ^NODE.
        nonterminal :: $\mathbb{N}$.
{ Returnstate is used during the parsing of a rhs; after recognition of the rhs returnstate is cleared; nonterminal is then filled by the nonterminal at the lhs (in case of a cf rule) or by the statenr (of QL) associated with the start of the lhs (in case of a cs rule).
If the grammarrule was cf then this infolist will serve as the parse of the reduced nonterminal. It is then called "infolist_symbol". If the grammarrule was cs this infolist will serve as an association list for the processing of cover symbols at the lhs and will be attached to a node in QR. Path is a (partial) parse. In variable_list the values of variables are represented. Lex_info is used during the processing of a lexicon symbol. }

LEX_INFO :: (place, is_an_entry).
        place :: a pointer in a trie-structured lexicon.
        is_an_entry :: $\mathbb{B}$.
{ We described the structure of a trie-structured lexicon on external memory in chapter 2. The structure of a pointer in such a lexicon ("place") is described in (Skolnik, 1982). The positions between characters in such a lexicon can be viewed as states and a "place" in a lexicon as a pointer to such a state. A "place" may be marked if from the start of the lexicon up to the place an entry can be spelled out. If that is the case the boolean "is_an_entry" will be set to true and the eventual information which is denoted with the entry in the lexicon is transferred to the variables in the infolist where lex_info is included. }

VARIABLE_LIST :: (refcount, set_of_variables)*.
        set_of_variables :: ^SET_OF_VARIABLES.
{ The set of variables may have multiple sets of values.}

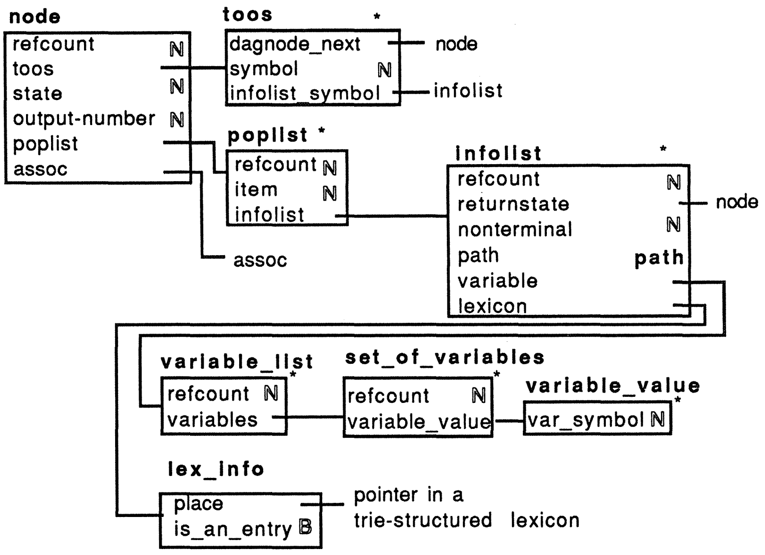
SET_OF_VARIABLES :: (refcount, variable_value)*.
        variable_value :: ^VARIABLE_VALUE.

{ A set_of_variables contains a list of values of variables. The variables are identified by the sequence in which they are denoted in a rule for the first time : during code generation this static sequence is known. }

VARIABLE_VALUE :: var_symbol .
    var_symbol :: (ℕ)*.
{ A value of a variable consists of a string of 0 or more characters, represented as integers. }

Graphical description of a node

(* means : a list of these elements)



4.2.1.2 Parse forests and association lists

Informal description
In chapter 3 we introduced a condensed path within a condensed parse forest. A path may also be used for the evaluation of cover symbols. This will be exemplified in an example of the positioning within a parse on the next leaf symbol.

Formal description

F :: (ACTION_HEADER)* .

ACTION_HEADER :: (refcount, chain, action_set, symbol, ^infolist_symbol, ^actionlist) .
    chain :: ^ACTION_HEADER .
    action_set :: ℕ*.
    symbol :: ℕ.
{ An "action_header" is a node of a parse forest. In "action_header_chain" the default pointer to the next action_header is stored; alternatives to "action_header_chain" are stored in the al-

ternate_chain lists which belong to an infolist. In "actionlist" a deep-binding is possible for the reports and builds. In action_set the union is stored of all the kind_of_action's which are present in the actionlist (for the purpose of speeding up the retrieval of certain actions). }

ACTIONLIST :: (refcount, kind_of_action, action_parts)* .
　　　　kind_of_action :: ℕ.
　　　　action_parts :: ℕ*.
{ An actionlist contains the binding of actions; the art of the binding is given in "kind_of_action" }

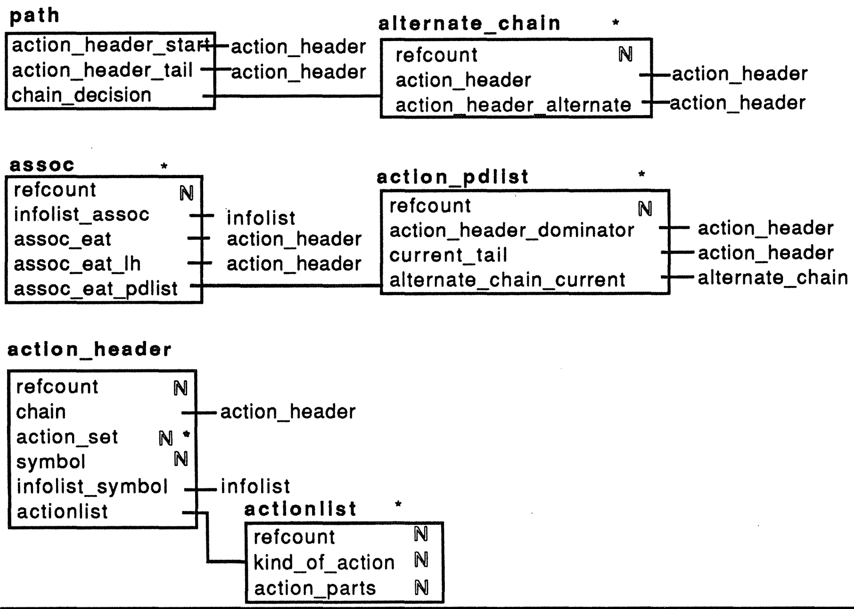
PATH :: (^action_header_start, ^action_header_tail, chain_decision).
　　chain_decision :: ^ALTERNATE_CHAIN.
{ Action_header_start and _tail point to the head and the tail of a condensed path in the forest, which represents the current parse; alternate_chain specifies the exceptions to the path from action_header_start to action_header_tail. }

ALTERNATE_CHAIN :: (refcount, ^action_header, ^action_header_alternate)* .
{ Alternate_chain serves to specify the exceptions within a condensed path. If in a path "^action_header" is reached then "^action_header_alternate" has to be chosen as its successor, rather then the "chain" which is noted with the "^action_header". See also the example. }

ASSOC :: (refcount, ^infolist_assoc, assoc_eat, assoc_eat_lh, assoc_eat_pdlist) * .
　　assoc_eat | assoc_eat_lh :: ^ACTION_HEADER .
　　assoc_eat_pdlist :: ^ACTION_PDLIST .
{The datastructure ASSOC provides for the binding of a nonterminal with its covered parse tree. Assoc will be nil for L-nodes; for R-nodes it contains in "infolist_assoc" the parse for the rhs of the rule belonging to them; in this parse nonterminals may be found "associated" with their covered parse tree. A pushdown list is maintained in order to read in sequentially for a cover symbol at a lhs of a rule its covered terminals. The positions in "assoc_eat_pdlist" are related to the position of the lookahead symbol. The current symbol is pointed to by "assoc_eat", the next symbol (necessary for look-ahead calculations) by "assoc_eat_lh". In the general case nonterminals may cover more parse trees. In that case infolist_assoc consists of more elements. "Assoc" in node_r_current will then contain more elements. See also the example.}

ACTION_PDLIST :: (refcount, ^action_header_dominator, current_tail,
　　　　　　　^alternate_chain_current)* .
　　current_tail :: ^ACTION_HEADER .
{ An "action_pdlist" is a pushdown list for the remembrance of the position in a parse. It is used in a node of the R-dag when input is taken, during "lazy evaluation", of the associated parse tree of a cover symbol. In "current_tail" is stored the tail belonging to the nonterminal dominating this level. In "alternate_chain_current" is stored the alternate_chain which belongs to the nonterminal dominating the current level. See also the example.}

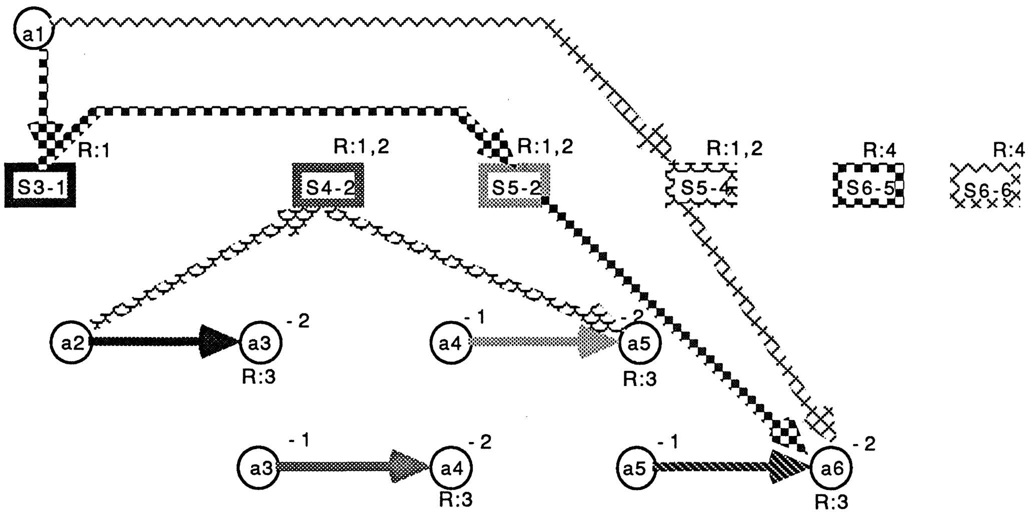
Graphical description.



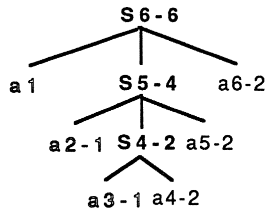
Example
In chapter 3 we gave an example of the creation of a parse forest during the reading of the input. We repeat the last figure, which represents the parse forest for two parses, together with one of the reconstructed parse trees.
Written in set-notation, the condensed parse forest is :
((a1, S3-1), (S3-1, S5-2), (S5-2, a6-2), (S5-4, a6-2), (a2-1, a3-2), (a3-1, a4-2), (a4-1, a5-2), (a5-1, a6-2), (S6-5, ε), (S6-6, ε), (S4-2, a5-2)). Of interest for our example are the following condensed paths : for S6-6 : ((a1, a6-2), (a1, S5-4)), for S5-4 : ((a2-1, a5-2), (a2-1, S4-2)) and for S4-2 : ((a3-1, a4-2), ε).

<u>Situation after reading 'aaaaaa)'</u>

Grammar: S'::S{R:4},).
S::a,[S{R:1}]*{R:2},a{R:3}.



The parse for S6-6 was :

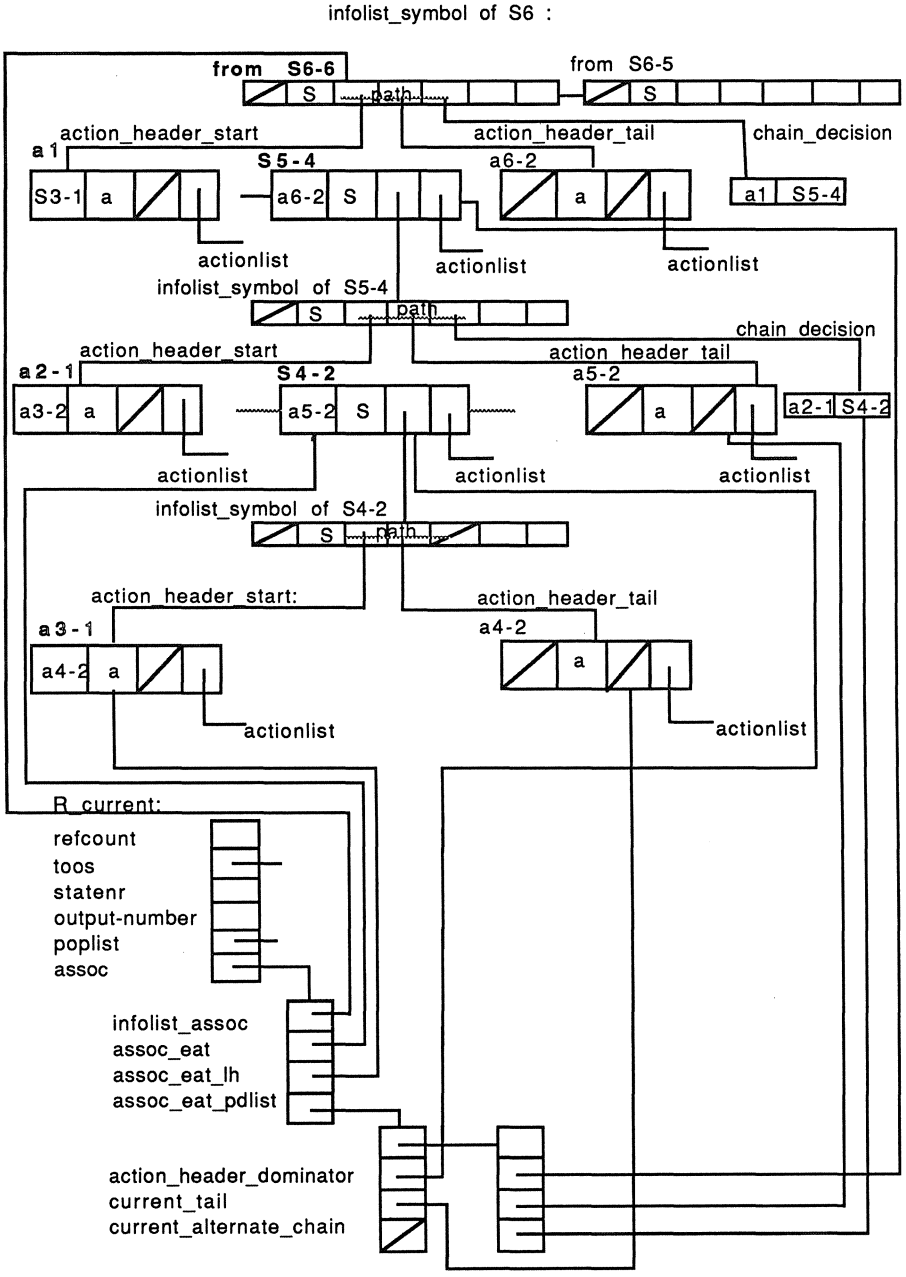


Consider the transduction grammar

a, S^ :: S {R:4},) .
S :: a, [S {R:1}]* {R:2}, a{R:3}.

which contains the same rhs's as the grammar of the example. We suppose also
- that 'aaaaaa)' is read from the input
- that we are transducing according to the code for the lhs of the first rule
- that we are treating the cover symbol S^
- that we are generating symbols for S^ according to the parse of S6-6 (in parallel symbols can be generated from the parse of S6-5, but we disregard this for the sake of this example)
- that we generated already 2 a's.
In that case the following relevant datastructures are instantiated :

Representation of the assoc for S6-6 after generating 'aa' and positioning on the lh symbol a3-1.



4.2.1.3 Structure of connectors

Informal description

In chapter 3 we already introduced the role of a connector. The configuration of a connector indicates which nodes in L and R cooperate and where the next input symbol has to come from.

In principle a connector is a 4-tuple (active nodes on L, projective nodes on L, active nodes on R, projective nodes on R). The names which are used in the description are respectively : "active_l", "local_projective_l", "active_r" and "projective_r". These names stem from their use in the shift instructions. active nodes project themselves, on the reading of an input symbol, to projective nodes. The number of possible projective nodes is limited by the number of compiled states in QL and QR.

For the purpose of the treatment of reduce instructions a few more elements are added. Because of the different treatment of cf and cs reduce instructions a subdivision is made. After a cf reduce a connector is made with "leftsymbol_cf" filled with the reduced symbol at the lhs. The infolist which belonged to the reducing item at the rhs is then transferred to "reduce_infolist" in the new connector. From the current connector are copied the local_projective_l, the active_r and the projective_r. This will be dealt with further in the discussion of the processing of connectors.

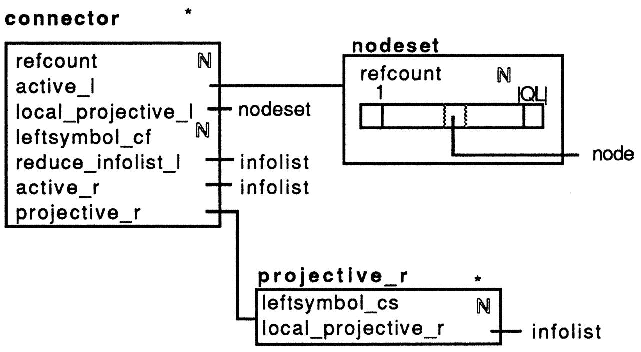
Formal description of a connector :

C :: (C_0, CONNECTOR*).
C_0 :: CONNECTOR.

CONNECTOR ::(refcount, active_l, local_projective_l, leftsymbol_cf, reduce_infolist_l, active_r, projective_r).
active_l | local_projective_l :: ^NODESET.
reduce_infolist_l | active_r :: ^INFOLIST.
projective_r :: (leftsymbol_cs, local_projective_r*)*.
local_projective_r :: ^INFOLIST.
leftsymbol_cf | leftsymbol_cs:: $\mathbb{N}$.

NODESET :: (refcount, $[\text{^NODE}]^{0..|QL|}$).

Graphical description of a connector.



4.2.2 Processors and programs

The processor PT for the PTA reacts on the input. The processor PC for connectors is activated by PP and reacts on the configurationconfiguration of a connector of a connector. The processors PL and PR for programs which are associated with states in QL and QR are in their turn activated by PC. The individual instructions in I are activated by PL and PR and react on parts of a configuration of the PTA. The resulting operation does not interfere with other operations which may be initiated in parallel.

4.2.2.1 PT : processor for the PTA

Informal description of PT.

PT operates in the environment in which the PTA is used. We may identify it with the process Gp into which a U-grammar G will be transformed by the compiler. In section 2.2 we specified the input and output of this process as

input	output
the input string (tree)	the lexicon
the lexicon	the result of the recognition : true or false
	the resulting parse
	the resulting output
	the resulting reports
	the resulting builds.

The PTA generates output by the pruning of internal datastructures (to be discussed in section 4.2.2.6). There are no instructions for the explicit creation of output. Therefore the output-parameters will not be passed further to the other processes.

Graphical description of PT.

Nothing can be shown : before and after the process the PTA consists solely of the base node L_0.

Procedural description of PT.

```
Program PT { processor for PTA }
- initialise input and output
- $L_0$ := (1, ∅, 1, 0, (0, 1, ∅), ∅)
- active_l := {$L_0$}
- proceeding_possible := true; accepted := false
- while not end-of-file and proceeding_possible do
   - read symbol_of_text
   - global_projective_l := ∅
   { a queue of connectors is maintained }
   - queue_of_connectors := ∅
   - for all L-nodes in active_l do
      - PL (node_lsymbol_of_text, global_projective_l, accepted)  { configuration A }
   - while queue_of_connectors <> ∅
      - take next connector
      - PC (active, global_projective_l, symbol_of_text, accepted)
      - transform connector
   - if global_projective_l = ∅
```

```
        - if transduction grammar
           - global_projective := {$L_0$}
        - else proceeding_possible := false
     - dispose active
     - active := global_projective
- result_of_recognition := accepted
{ end of PT }
```

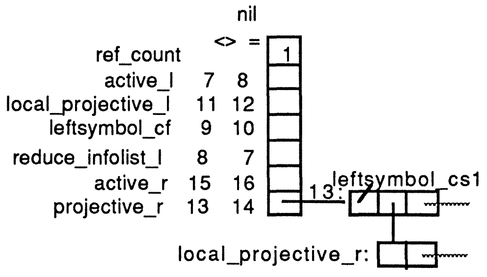
4.2.2.2 PC : processor for connectors

Informal description of PC.

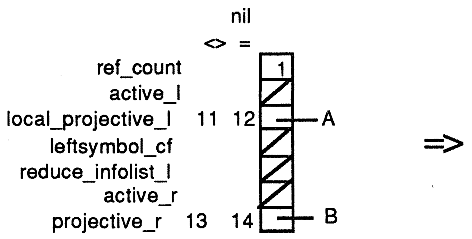
Some configurations of a connector give rise to the activation of processors PL and PR. After these processors have finished the configuration changes. It is possible that the connector reaches a configuration which is not in a form that allows immediate further action of a processor. We call this an instable configuration. In that case the connector brings itself into an allowed configuration or it disappears. An allowed configuration may again give rise to the activation of processors, etcetera.

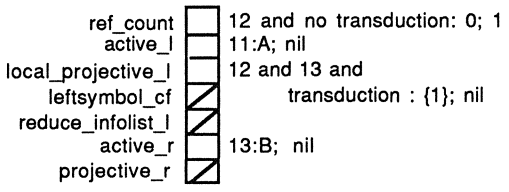
Graphical description of PC.

General configuration of a connector on which processing is based :



Transitions of instable configurations of connectors





(a refcount of 0 implies pruning)
12 and 14 and transduction : global_projective_l := {1}

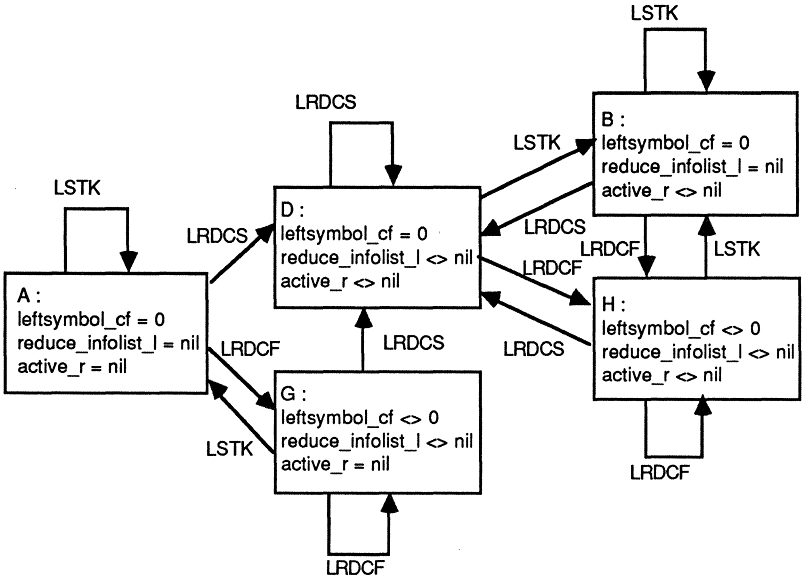
Non-procedural description of PC.

The possible configurations of a connector, together with the source of the next input symbol and the resulting activation of the processors PL and PR are shown in the following table:

	left sym bol_cf	reduce info list_l	acti ve_r	projec tive_r	action :		
					symbol from	proces sor	works on nodes in
A	0	nil	nil	nil	input	PL	active_l
B1	0	nil	<>nil	nil	- -	PR	active_r
B2	0	nil	nil	<>nil	projective_r	PL	active_l
C	0	<>nil	nil	impossible: with a reduce_infolist_l a leftsymbol or a leftitemlist or both have to be present			
D1	0	<>nil	<>nil	nil	- -	PR	active_r
D2	0	<>nil	nil	<>nil	projective_r	PL	reduce_infolist_l
E	<>0	nil	nil	impossible : a leftsymbol_cf needs a reduce_infolist_l			
F	<>0	nil	<>nil	impossible: see E			
G,H	<>0	<>nil	*	nil	leftsymbol_cf	PL	reduce_infolist_l

After the processing of a set of nodes its reference in the current connector is cleared.

During the processing of a connector new connectors may be created by instructions which operate on the L-dag. As a reference we indicate in a finite-state diagram the configuration of a connector which results from the execution of an instruction in the context of a connector. This reference can be reconstructed from the formal definition of the instructions.
For each configurationconfiguration of a connector A, B, D, G, H of a connector we give, for each instruction LSTK (shift on the L-dag), LRDCF (cf reduce on the L-dag), LRDCS (cs reduce on the L-dag) the resulting connector.



Procedural description of PC.

```
Procedure PC (nodeset_active, nodeset_projective, symbol_of_text, accepted)
  - if leftsymbol_cf <> 0 { connector constructed because of reduction of a cf rule }
    - leftsymbol_cf_infolist := copy of reduce_infolist_l with returnstate cleared
    - for all L-nodes in reduce_infolist of connector do
      - if active_r = nil
        - PL (node_l, leftsymbol_cf, leftsymbol_cf_infolist,
                  global_projective_l, accepted ) { configuration G }
      - else { active_r <> nil }
        - local_projective_l := ∅
        - PL (node_l, leftsymbol_cf, leftsymbol_cf_infolist,
                  local_projective_l, accepted )  { configuration H }
        - add to queue_of_connectors a new connector with
          - active_l := local_projective_l
          - active_r := active_r of current connector
  - else
    - for all node_rs in active_r {of connector} do
      - PR (node_r, projective_r) { execute program instructions for node_r.state
            and construct a projective_r, which contains all acting
            leftsymbol_cs's and their local_projective_r }
    - for all leftsymbol_cs in projective_r do
      - for all R-nodes in the local_projective_r of leftsymbol_cs do
        - if reduce_infolist_l {of connector} <> nil
          { connector constructed because of reduction of a cf rule }
          - local_projective_l := ∅
          - for all L-nodes in reduce_infolist_l {of connector} do
            - PL (node_l, leftsymbol_cs from projective_r, local_projective_l, accepted )
              { configuration D }
          - if no inhibition by a Boolean expression of variables
```

```
                - add to queue_of_connectors a new connector with
                  - active_l := local_projective_l
                  - active_r := local_projective_r of projective_r
            - else {reduce_infolist_l = nil, the connector was created by a push instruction}
              - if local_projective_r <> nil {R stack still busy}
                - create local_projective_l
                - for all L-nodes in active_l do
                  - PL (node_l, leftsymbol_cs from projective_r, local_projective_l,
                        accepted )
                    { configuration B }
                - if no inhibition by a Boolean expression of variables
                  - add a connector to queue_of_connectors with
                    - active_l := local_projective_l
                    - active_r := local_projective_r of projective_r
              - else {local_projective_r = nil, R stack empty, connect now to
                        global_projective_l}
                - for all L-nodes in active_l do
                  - PL (node_l, leftsymbol_cs from projective_r, global_projective_l,
                        accepted ) { towards configuration A }
{ end of PC }
```

4.2.2.3 PL and PR : processors for nodes in L and R

Informal description.
The processors PL and PR are activated from a connector. PL will operate on the program code for a state in QL. The program code for all states in QL is described by "code_rhs" . PR will operate on the program code for a state in QR. The program code for all states in QR is described by "code_lhs".
The program code in total is described by "code". In the subroutines the code is sampled for the sub-formalisms which are denoted between the action-brackets and which are treated by the compiler.

Non-procedural description.

```
code          :: [ reporting_subroutine ]* , [ variable_subroutine ]* ,
                 [ receive_subroutine | send_subroutine , [ closure_subroutine ] ]* ,
                 [ code_lhs ] , code_rhs .
```

4.2.2.3.1 PL : processor for nodes in L

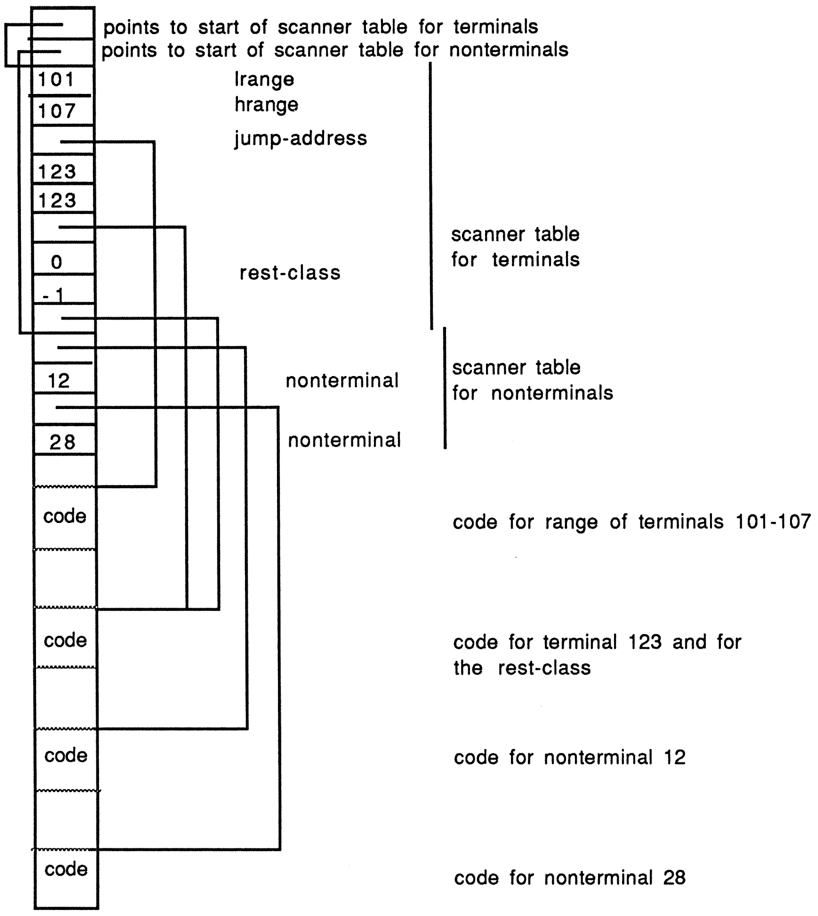
Informal description of PL.
PL processes a program which is associated with a compile-state for the L-dag and a symbol. The allowed sequence of instructions is described by a cfg.
"Right_code" describes for each compile-state the code ("rightstate_code"). This code is pointed to by two tables, one for terminals and one for nonterminals and intermediate symbols which constitute the "scanner-table". The first table consists of triplets (lrange, hrange, jump_address). If lrange <= symbol <= hrange then the code_per_symbol is selected which is addressed by jump_address. The second table consists of duplets (nonterminal_nr, jump_address). This table will be selected when the input-symbol is not a terminal.

During the processing of code_per_symbol the instructions have access to the parameters which are supplied in the configuration ("PARAMS"). These parameters may be altered by the instructions. The instructions themselves may have parameters which are provided within the code. In the following table we mention the PARAMS- and instruction-parameters.

	parameters within PARAMS :					
instruction with paramater(s)	^node_l curren	items	symbo	^infolist_ symbol	^infolist_former _and_current	node_l projective
TOP item_cur,item_proj	in	out	in	in	out	
LSTK statenr_l	in		in	in	in	out
PLRDC leftsymbol_cf	in, out		in	in	in, out	
LRDCF leftsymbol_cf				in		
LRDCS statenr_r				in		
SRL reportnr				in , out		

The instructions within l_state_code start with an optional instruction LSTK and an obliged instruction TOP. The last one has as a parameter the current item. With this item is stored an infolist with references to a condensed path in the forest and references to variables and the lexicon. The results of succeeding operations on the forest, the variables and the lexicon are stored according to these references. After these operations the shift and reduce instructions are executed on the L- and R-dag.

Graphical description of PL.
It is not appropriate to show the sequential organization of PL in a graphical form. But we will show the organization of the scanner table by means of an example.



points to start of scanner table for terminals
points to start of scanner table for nonterminals
101
107
lrange
hrange
jump-address
123
123
0
-1
rest-class
scanner table for terminals
12
nonterminal
28
nonterminal
scanner table for nonterminals
code
code for range of terminals 101-107
code
code for terminal 123 and for the rest-class
code
code for nonterminal 12
code
code for nonterminal 28

Non-procedural description of PL.

```
! subroutines for reporting !
 reporting_subroutine:: [ report_code ]+ , RTN .
 report_code          :: SRL, reportnr .

! subroutines for operations on variables !
 variable_subroutine :: [ assignment ; equal_test ; inequal_test ]+ , RTN .
 assignment          :: expression, ASS , variable_name .
 equal_test          :: left_expression, right_expression , TEQ .
 inequal_test        :: left_expression, right_expression , TNE .
 left_expression     :: expression .
 right_expression    :: expression .
 expression          :: PSH , varorlit_name , [ CAT , varorlit_name ]* .

! subroutines for transport of variables!
 send_subroutine     :: ALL , n_variables , [ SND , variable_name ]* , RTN.
 closure_subroutine  :: [ CLI , closure_item , send_call ]+ , RTN .
```

```
! subroutines for lexicon interface !
  lex1_subroutine     :: [ LEXST | LEXINC ]1, RTN.
  lex2_subroutine     :: LEXINC, LEXRDC, RTN.
  receive_subroutine  :: [ RCV , variable_name ]+ , RTN .

! code for L !
  code_rhs            :: [ l_state_code ]+.
  l_state_code        :: [ leftsymbol_subroutine | list_subroutine ]* ,
                             [ code_per_symbol ]+ , jump_list , 0 .
  jump_list           :: terminal_jump_list, [ nonterminal_jump_list ] |
                         nonterminal_jumplist .
  terminal_jump_list  :: [ lrange , hrange , jump_address ]+.
  nonterminal_jump_list      :: [ jump_address , [ nonterminal_nr ]+ ]+.

  code_per_symbol     :: [ LSTK , statenr_l ] , [ list_of_instructions ]* , EXT |
                         NSTK , statenr_l, EXT .
  list_of_instructions :: TOP , from_goto_action , [ closure_call ] , [ instruction ] ,
                             [ list_of_instructions ]*  |
                         TOP , from_goto_action , leftsymbol_call, |
                         leftsymbol_call  | list_call.

  from_goto_action    :: from_goto_pair , [ reporting_call ]*  |
                         closure_item_current, item_projective , lex1_call |
                         item_current, item_projective , lex2_call .
  from_goto_pair      :: closure_item_current , item_projective , [ ALL , n_variables ] ,
                             [ [variable_call ]* , receive_call ] |
                         item_current , item_projective ,  [ receive_call ] .
  instruction         :: PLRDC, leftsymbol_cf   | LRDCF, leftsymbol_cf   | LRDCS  ,
                             statenr_r  | ACC .
  list_subroutine     :: [ list_of_instructions ]+ , RTN.
  leftsymbol_subroutine        ::  PLRDC , leftsymbol_cf , [ list_of_instructions ]+ , RTN .

  leftsymbol_call | list_call | action_call | closure_call | variable_call | reporting_call |
        send_call | receive_call | lex1_call | lex2_call          :: CAL , jump_address .

  jump_address | item_current | item_projective | reportnr | statenr_r | statenr |
        leftsymbol_cf | nonterminal_nr  | variable_name | n_variables  :: pos_integer.
  closure_item_current         :: neg_integer .
  lrange | hrange              :: 0..255.
  varorlit_name                :: variable_name | 0 ! if symbol last-read! |
                                        neg_integer ! if literal  ! .
```

Procedural description of PL.

The procedural description of the processor PL follows the grammar for l_state_code in a straightforward manner. A program for PL can be derived also in an automatic way by generating a recognizer for this grammar.

Procedure **PL** (node_l_current, symbol, infolist_symbol, projective_l, accepted)
- program_counter := scanner(node_l_current.statenr, symbol)
- command := program_code(program_counter)

```
- if command = LSTK
  then - projstatenr_l := program_code(program_counter+1)
       - push(node_l_current, node_l_projective, statenr_l_projective, projective_l, symbol,
                        infolist_symbol )
       - program_counter := program_counter+2
- exit := false
- while not exit do
   - command := program_code(program_counter)
   - case command of
      - CTOP : TOP :  - LIST_OF_INSTRUCTIONS(node_l_current, node_l_projective,
                        symbol, infolist_symbol, accepted)
      - CAL :          - push_address(program_counter+1)
                       - program_counter := program_code(program_counter+2)
      - RTN :          - program_counter := pop_address
      - EXT :          - exit := true
{ end of PL }
```

Procedure LIST_OF_INSTRUCTIONS(node_l_current, node_l_projective, symbol, infolist_symbol, accepted)

```
{ the program_counter is positioned at the instruction (C)TOP }
- item_current_is_a_closure_item := program_code(program_counter) = CTOP
- item_current := program_code(program_counter+1)
- item_projective := program_code(program_counter+2)
- infolist_former_and_current := PF(node_l_current, item_current, program_counter)
- command := program_code(program_counter+2)
- while command = CAL
   - push_address(program_counter+2)
   - program_counter := program_code(program_counter+1)
   - command := program_code(program_counter)
- case command of
   - PLRDC:        - leftsymbol_cf := program_code(program_counter+1)
                   - PSEUDO_CF_REDUCE(leftsymbol_cf,
                           infolist_former_and_current);
                   - program_counter := program_counter+2
   - LRDCF:        - leftsymbol_cf := program_code(program_counter+1)
                   - CF_REDUCE(leftsymbol_cf, infolist_former_and_current,
                           connector.projective_l, local_projective_r)
                   - program_counter := program_counter+2
   - LRDCS:        - statenr_r := program_code(program_counter+1)
                   - CS_REDUCE(statenr_r, infolist_former_and_current,
                           connector.projective_l, local_projective_r)
                   - program_counter := program_counter+2
   - LEXST:        - initialise the lexiconpointer of infolist_former_and_current on the
                           start of the lexicon
   - LEXINC:       - increment the lexiconpointer of infolist_former_and_current
   - LEXRDC:       - create a connector with an infolist which contains in the variables
                           categories from the lexicon
   - ACC:          - ACCEPT(infolist_former_and_current)
                   - accepted := true
                   - program_counter := program_counter+1
   - otherwise:    - PUSH_ITEM(node_l_current, node_l_projective,
                                   item_current, item_projective, infolist_former_and_current)
- while program_code(program_counter) = RTN
    - program_counter := pop_address
- infolist_former_and_current.refcount := 0
{ end of LIST_OF_INSTRUCTIONS }

Procedure PSEUDO_CF_REDUCE (leftsymbol_cf, infolist_symbol)
- denote leftsymbol_cf in all elements of infolist_symbol
  { because all elements of infolist_symbol have to be treated with the same code we
    remember the program_counter }
- program_counter_saved := program_counter
- while infolist_symbol <> ∅ do
    - node_1_current := infolist_symbol.returnstate { perform the remaining instructions
          as if we were in the returnstate }
    - infolist_symbol.returnstate := ∅ { the returnstate is not longer needed }
    - program_counter := program_counter_saved
    - exit := false
    - while not exit do
        - command := program_code(program_counter)
        - case command of
            - CTOP : TOP :     - LIST_OF_INSTRUCTIONS (node_1_current,
                                       node_1_projective, symbol, infolist_symbol, accepted)
            - CAL :            - push_address(program_counter+1)
                               - program_counter := program_code(program_counter+2)
            - RTN :            - program_counter := pop_address
            - otherwise :      - exit := true
    - save_next := infolist_symbol.next
    - infolist_symbol.refcount := 0 { this element of infolist_symbol may be pruned }
    - infolist_symbol := save_next
{ end of PSEUDO_CF_REDUCE }
```

4.2.2.3.2 PR : processor for nodes in R

Informal description of PR.

Analogous to PL, the processor PR processes instructions for the R-dag. In principle the two processors can be the same. However, because of the simpler formalism at the lhs of a rule in a U-grammar PR can be simplified.

PR processes a program which is associated with a compile-state for the R-dag and a symbol. The difference with PL is that the symbol is not read from input nor originates from a cf reduction (and is compared with the symbols in the code) but that the symbol will be provided by the code (with the instruction RGTS, or, in the case of a cover symbol, by an assoc list). The allowed sequence of instructions is described by a cfg for "code_rhs". It contains the code ("r_state_code") for each compile-state.

During the processing of r_state_code 3 global references to substructures are maintained. They may be altered by the instructions. The instructions themselves may have parameters which are provided within the code. In the following table the global- and parameter-variables are mentioned.

parameters within Params :

instruction with parameter(s):	node_r_ current	projective_r_ elem
RGTS leftsymbol_cs	in	out
RGTSC leftsymbol_cs	in	out
RSTK statenr_r_projective	in	in, out
RRDC	in	in, out

Non-procedural description of PR.

```
code_rhs      :: [ r_state_code ]+ .
r_state_code:: RGTSC, symbol | leftgotolist | leftreducelist .
leftgotolist  :: RGTS, leftsymbol_cs, RSTK, statenr_r .
leftreducelist:: RGTS, leftsymbol_cs, RRDC .
symbol        :: ascii_value | nonterminal_nr .
```

Procedural description of PR.

```
Procedure PR (node_r, projective_r)
- exit := false
- if node_r.assoc.assoc_eat <> ∅
   - leftsymbol_cs := next leave symbol of node_r.assoc
   - add to projective_r a projective_r_elem1 with leftsymbol_cs
         and as local_projective_r an infolist with node_r
   - exit := true
- while not exit
   - command := program_code(program_counter)
   - case command of
      - RGTS : - leftsymbol_cs := program_code(program_counter+1)
               - add to projective_r a projective_r_elem1 with only leftsymbol_cs filled in
               - program_counter := program_counter + 2
      - RGTSC : - nonterminal := program_code(program_counter+1)
               - lookup nonterminal in assoc.infolist_assoc and return its infolist_symbol
               - position assoc_eat on the first leave in infolist_symbol
               - program_counter := program_counter + 2
      - RSTK : - projstate_r := program_code(program_counter+1)
               - create a new node_r
               - add an element to the local_projective_r of projective_r_elem1
                     with node_r filled in
               - program_counter := program_counter + 2
      - RRDC : - add poplist of node_r to local_projective_r
               - program_counter := program_counter + 1
      - EXT :  - exit := true
{ end of PR }
```

4.2.2.4 PF : processor for the parse forest

Informal description of PF.
The processor PF processes instructions which have an effect on the construction of the parse forest. These instructions are usually generated when a parse tree has to be built, when an association list has to be kept for cover symbols or when static information has to be stored temporarily in the forest, waiting for the pruning processor PP to release them to an output stream.

Procedural description of PF.

```
Infolist function PF(node_1_current, item_current, program_counter)
- if item_current present
   - infolist_former_and_current := the infolist of the poplist of item_current
                                          in node_1_current
- else (no item_current, so a closure-item which was not already present)
   - infolist_former_and_current := a new infolist with node_1_current as returnstate, rest nil
- program_counter := program_counter+3
- new_actionheader := ACTIONS_FROM_PROGRAM(program_counter,
                                   symbol, infolist_symbol)
- ATTACH_ACTIONHEADER_TO_INFOLIST (new_actionheader,
                                   infolist_former_and_current)
{ end of PF }

Actionheader function ACTIONS_FROM_PROGRAM(program_counter,
        new_action_header, symbol, infolist_symbol)
- if action-instructions are present, or if transduce or if infolist_symbol <> nil
        or if build_parse tree
   - create new_action_header with symbol and infolist_symbol
   - while program_code[program_counter] = SRL
      - reportnumber := program_code(program_counter+1)
      - ADD_ACTION(new_action_header, reportnumber)
      - put kind_of_action (= report, or other) in the set
               new_action_header^.kinds_of_actions
      - construct an actionlist-element, pointed to by new_actionlist,
               with the action-information
      - insert new_actionlist in new_action_header^.actionlist
      - program_counter := program_counter+2
      - actions_from_program := new_action_header
- else actions_from_program := nil
{ end of actions_from_program }

Infolist function ATTACH_ACTIONHEADER_TO_INFOLIST (new_actionheader,
        from_infolist)
- make a copy of the infolist, pointed to by from_infolist, and call it
        infolist_former_and_current
        (the chain_decisionlist need not to be duplicated, only the pointer to it)
- if new_actionheader <> NIL  (new actions)
   - for all tail_action_headers in the elements of infolist_former_and_current do
      - if tail_action_header^.chain = nil
         - tail_action_header^.chain := new_actionheader
        - else
           - construct an alternate_chain with action_header := tail_action_header
                  and alternate_chain := new_action_header
           - make a copy of the list pointed to by element^.chain_decision and
                  call it copy_chain_decision
           - element^.chain_decision := copy_chain_decision || alternate_chain
        - tail_action_header := new_actionheader
- (if no new action there is nothing to change in infolist_former_and_current)
  attach_actionheader_to_infolist := infolist_former_and_current
{ end of attach_actionheader_to_infolist }
```

4.2.2.5 Processors for individual instructions in PL, PR and PF

Informal general description.
The instructions in set I operate in an environment provided by the nodes which are referenced by a connector.
The working of an instruction is described by the rewriting of a configuration:
CONFIGURATION2 :: CONFIGURATION1 .

The description-grammar for a configuration is :

```
        CONFIGURATION :: (connector*, L, R, PARAMS).
        PARAMS :: ^NODE, item_current, item_projective, symbol, ^infolist_symbol,
                        current parse.
        item_current | item_projective | symbol :: ℕ.
        current_parse :: ^INFOLIST.
```

The whole context for the applicability of an instruction is part of a configuration. The relevant parts of the context are specified by writing patterns.

Instructions are grouped within a program.The allowed sequences of instructions within a program are described by the grammars which are denoted with the processors PL and PR. Instructions will usually operate in the context of other instructions, as described in the grammar. The context is then passed over by the parameters in PARAMS.

In chapter 3 we recapitulated the 4 instructions which may be contained in a LR-table: shift, reduce, accept and error. In chapter 5 and 6 we will show how the LR-table construction algorithm can be extended for the unifying formalism in order to generate the following instructions.

There are 3 instructions which only effect the execution order of the code :

CAL , jump_address	call a subroutine at jump_address; a subroutine contains code which, because of space-efficiency, is shared
RTN	return from a subroutine
EXT	exit the code for a compiled state

There are 3 instructions which only effect the (concurrent) building of parse trees and the storage of information in it :

(C)TOP, item_current, item_projective	proceed with alternative paths of code: get current parse for item_current; CTOP indicates that item_current is a starting item

PLRDC, leftsymbol_cf	"pseudo-reduce of a symbol at a lhs" : put leftsymbol_cf on top of the current parses
SRL , reportnr	"supply a report to an actionlist" : add reportnr to parse.

Similar instructions like SRL may be supported, like the build instruction or other output instructions which store their result temporarily in the parse forest, where they wait for their final release to their associated output-stream by the pruning processor PP.

There is 1 instruction which signals the acceptance of a sentence :

ACC	accept

There are 2 instructions which provide a symbol from a lhs :

RGTS, leftsymbol_cs	get a symbol from the R-dag
RGTSC, leftsymbol_cs	get a symbol from the R-dag, this symbol is a cover symbol

There are 6 instructions which react on a symbol originating from the input-structure :

LSTK, statenr_l	shift on the L-dag
RSTK, statenr_r	shift on the R-dag
LRDCF, leftsymbol_cf	cf-reduce on the L-dag
RRDC	reduce on the R-dag
LRDCS, statenr_r	cs-reduce on the L-dag
NSTK, statenr_l	shift, but without the creation of a new node (with this instruction it is possible to simulate a FSA)

There are 5 instructions with which operations on variables are executed :

PSH variable_name	push variable or literal onto the stack
CAT variable_name	concatenate a variable with topmost element; replace top by resulting element
ASS variable_name	assign topmost element to variable
TEQ	test equality of 2 topmost elements
TNE	test inequality of 2 topmost elements

There are 4 instructions with which the transport of variables to closure items and from reduce items are executed :

ALL nr_of_variables	reserve room for (integer) number of variables
RCV variable_name	receive the value of a variable from an infolist_symbol in the PTA
SND variable_name	send the value of a variable to the formal parameter of a closure item
CLI integer	create a starting item with this number

There are 3 instructions for the handling of a trie-structured lexicon :

LEXST	initialise the search for an entry in the lexicon at the start of the lexicon
LEXINC	increment the pointer in the lexicon
LEXRDC	perform a reduce after an entry is found in the lexicon

There are 2 instructions for skipping in the input :

SKPTR, item_current, item_projective	skip in the input up to the ")" of the current tree-level
SKP integer	skip forward (integer > 0) or backward (integer < 0) in the input

These skip-instructions will not be discussed further.

Miscellaneous instructions CAL , RTN, EXT and ACC

Informal description.
The program code for a symbol in a compiled state is ended by the instruction EXT. The program code may contain subroutines which are called by the instruction CAL and which are ended with the instruction RTN. The instruction CAL has as its argument an address in the code. The instruction ACC signals the acceptance of an input. It is up to the implementer which action has to be taken on this instruction : to continue with alternative recognition/parse paths or to stop processing. In the case of transduction grammars no ACC instruction is present in the code.

Non-procedural description.
It is impractical to describe the instructions CAL , RTN, EXT in a non-procedural way. On this low level of processing we assume sequential processing. The instruction ACC signals a possible end of all processing within the PTA.

Procedural description.
The procedural description of these instructions is contained in the procedural description of PL.

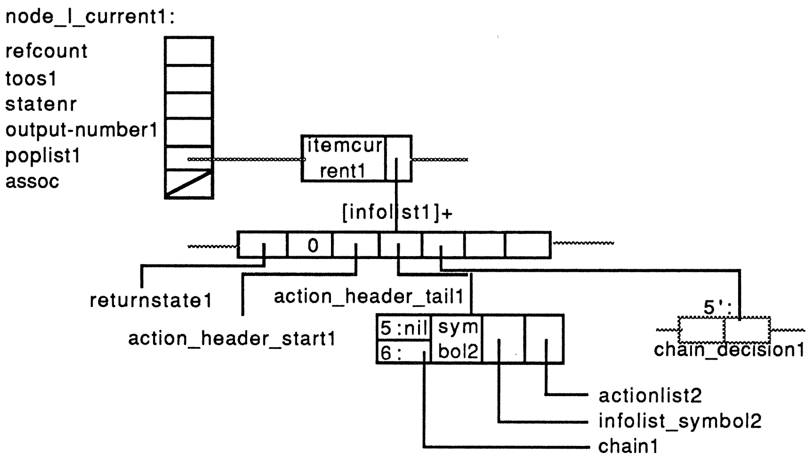
Instruction (C)TOP

Informal description.

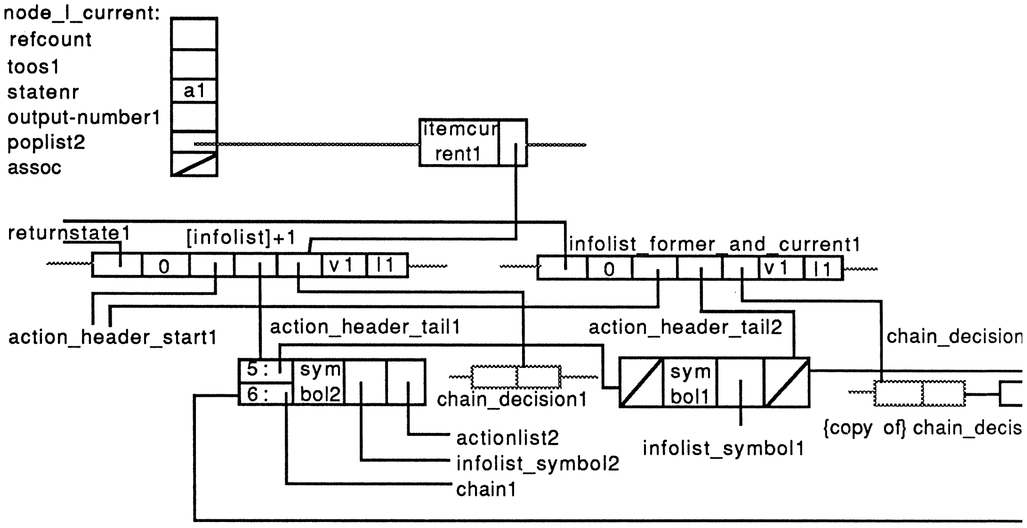
The instruction TOP precedes other instructions in order to provide for a reference to a partial parse. Is has 2 arguments : item_current and item_projective.
From the current runstate it takes (all) the infolist(s) which belong(s) to item_current and adds to it (them) an actionheader with the current symbol and its associated infolist_symbol in order to extend the parse forest. If there is more than one infolist then the actionheader will be connected to the tails of all the infolists. After the instructions TOP and CTOP instructions for actions may follow which have to be attached to the created actionheader. If a non-null item_projective is provided then a instruction LSTK has to follow. (There may follow many reduce instructions but only one instruction LSTK.)
The instruction CTOP operates like TOP, but item_current is a closure-item and will not be present in the poplist of node_1_current.

Graphical description.

Before instruction : TOP item_current1 item_projective1
Params : in: node_l_current1, symbol1, ^infolist_symbol1;
out: item_current1, item_projective1, ^infolist_former_and_current1.
configuration1 :: (connectors1,L1,R1).



After instruction :
configuration2 : (connectors1,L2,R1).



Non-procedural description.

```
Before instruction : TOP item_current1 item_projective1
Params1 : in: ^node_1_current1, symbol1, ^infolist_symbol1;
          out: item_current1, item_projective1, ^infolist_former_and_current1.

configuration1 :: (connectors1, L1, R1).
        L1 :: (-, node_1_current1, -).
        node_1_current1 :: ( -, ^poplist1, -).
        poplist1 :: (-, (item_current1, I), -).
        I :: [^infolist1]+.
        infolist1 :: (-, ^action_header_tail1, -) .
        action_header_tail1 :: ( [ 5: 0 | 6: chain1]1, -) .

configuration2 :: (connectors1, L2, R1).
        L2 :: (-, node_1_current2, -).
        node_1_current2 :: ( -, ^poplist1, -).
        poplist1 :: (-, (item_current1, I:[infolist2]+), -) .
        infolist2 :: (-, ^action_header_tail2, -) .
        action_header_tail2 :: ( [5: action_header_tail2 | 6: chain1]1, -) .
        infolist_former_and_current1 :: I:(-, ^action_header_tail2, -)+ .
        chain_decision2 :: chain_decision1, [ 6: (^action_header_tail1,
                                  ^action_header_tail3)]1 .
        action_header_tail3 :: (0, 0, symbol1, ^infolist_symbol1, 0) .
```

Procedural description.

The procedural description is contained within the processor PF.

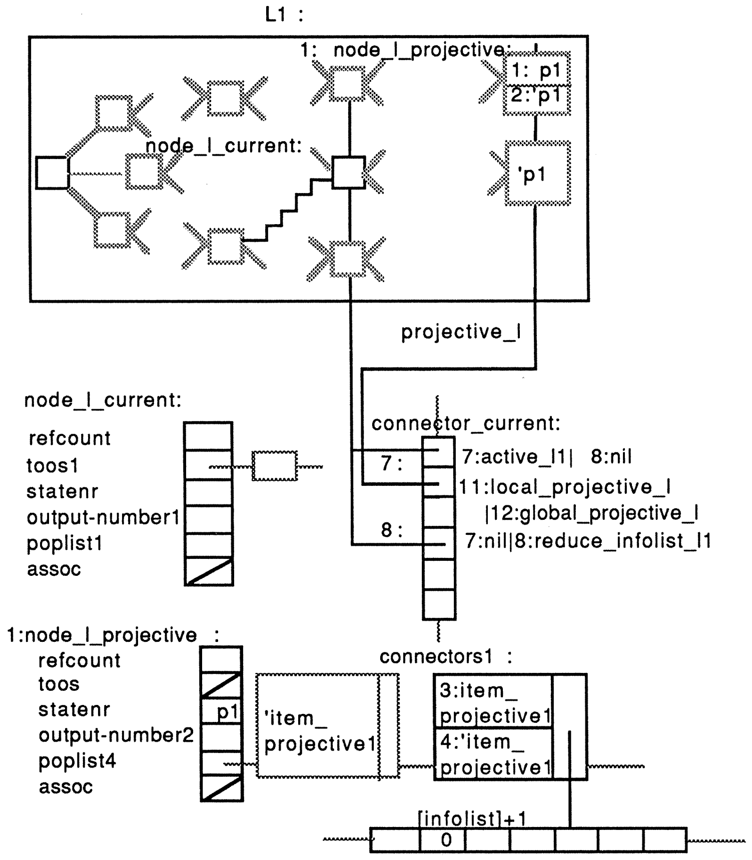
Instruction LSTK

Informal description.

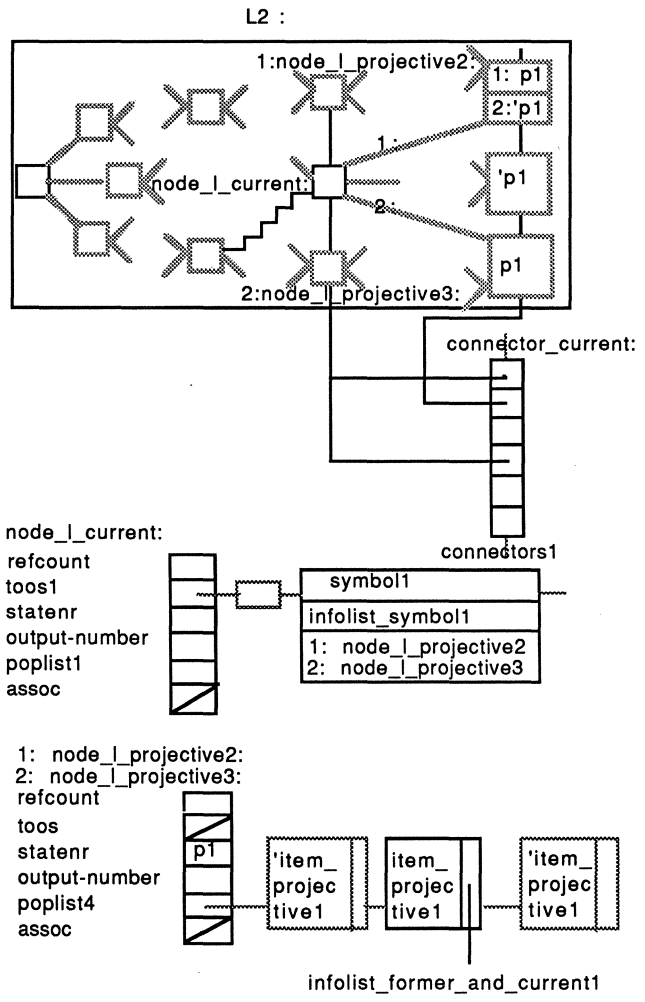
The instruction LSTK, with parameter 'statenr', performs a push on the L-dag from a runstate in an active nodeset to a runstate in a projective nodeset with compile-number 'statenr'. If such a runstate is not present then it has to be created.

Graphical description.

Before instruction : LSTK statenr_l_projective (=p1)
Params : in: ^node_l_current, symbol1, ^infolist_symbol1,
^infolist_former_and_current1; out: ^node_l_projective .
configuration1 :: (connectors1,L1,R1).



After instruction:
configuration2 : (connectors1,L2,R1).



Non-procedural description.

Before instruction : LSTK statenr_l_projective (=p1)
Params1 : in: ^node_l_current, symbol1, ^infolist_symbol1,
^infolist_former_and_current1; out : ^node_l_projective1.

configuration1 :: (connectors1, L1, R1).
L1 :: (-, node_l_current, -, [node_l_projective1], -).
connectors :: (-, connector_current, -).
connector_current :: (-, local_projective_l, -).
local_projective_l :: ([1: -, ^node_l_projective1, - | 2: [(0, p1', output-number1, ^poplist, 0)]*$1]1).

```
        node_l_current1 :: ( toos1, -).
        node_l_projective1 ::  (-, ^poplist4, -).
        poplist4 :: [ 3: popelements1, (item_projective1, [^infolist]+$2),  popelements2 |
                    4: popelements1]1.
        popelements :: [(p1', ^infolist)]* .

configuration2 :: (connectors1, L2, R1).
        L2 :: (-, node_l_current1, -, [ 1 : node_l_projective2 |  2 : node_l_projective3]1, -) .
        node_l_current1 :: ( toos2,  -).
        toos2 :: toos1, ([ 1: ^node_l_projective1 | 2 : ^node_l_projective3]1,
                        symbol1, ^infolist_symbol1) .
        node_l_projective2 :: (-, ^poplist5, -) .
        poplist5 :: popelements1, (item_projective1, [3: $2,
                        ^infolist_former_and_current1),  [3:popelements2]1 .
        node_l_projective3 : (0, p1, 0, ^poplist3, 0) .
        poplist3 :: (item_projective1, ^infolist_former_and_current1) .
        local_projective_l :: ( [1: -, ^node_l_projective1, -  |  2: $1, ^node_l_projective3 ]1 ).
```

Procedural description.

```
Procedure PUSH(node_l_current , node_l_projective, statenr_l_projective, projective_l,
                    symbol, infolist_symbol )
- if node_l_projective not already present in projective_l : create one
- if link not already present: create link between node_l_current and node_l_projective
{ end of push }
```

The execution of the following procedure is implied by the processor PL when the instruction LSTK is executed :

```
Procedure PUSH_ITEM(node_l_projective, item_current, item_projective,
                            infolist_former_and_current)
- if item_projective present in node_l_projective
   - get from node_l_projective the reference to the infolist of the poplist of item_projective;
        call it popinfo
- else (no item_projective)
   - construct a poplist-element for item_projective in node_l_projective; its infolist (reference
        popinfo) is nil
- merge infolist_former_and_current into popinfo
{ end of push_item }
```

Instruction PLRDC

Informal description.

"Pseudo left symbol" : complete a partial sub-parse tree by filling in the reducing leftsymbol_cf and removing runstate. Replace in Params the current runstate by the returnstate. This instruction is used during the building of a parse tree when a unit reduction is performed.

Graphical description.

Not necessary.

Non-procedural description.

PLRDC leftsymbol_cf1.
Params: in: ^node_l_current, symbol1, ^infolist_symbol1, ^infolist_former_and_current1.
out: returnstate1, leftsymbol_cf1, ^infolist_former_and_current1.

Before instruction:
infolist_former_and_current1 :: (returnstate1, 0, 0, -).

After instruction:
infolist_former_and_current1 :: (0, leftsymbol_cf1, ^infolist_symbol1, -).

Procedural description.

The procedural description is contained within the processor PL.

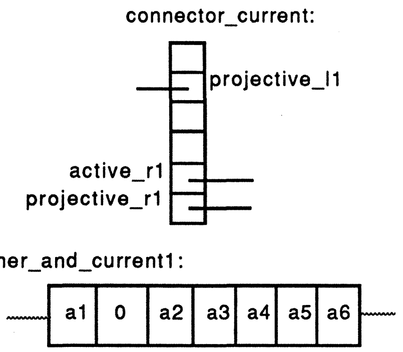
Instruction LRDCF

Informal description.

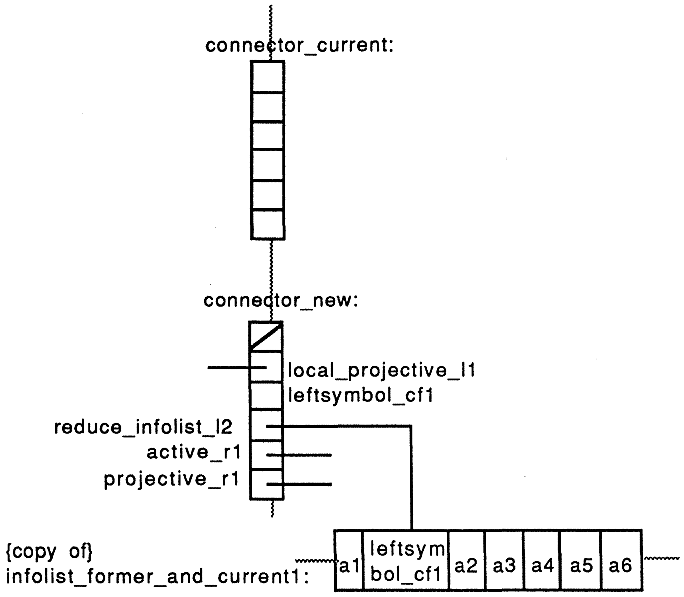
This instruction performs a cf reduce on the L-dag. It creates a new connector with infolist_former_and_current which provides for the returnstates and the associated parse trees.

Graphical description.

Before instruction : LRDCF leftsymbol_cf1
Params : in: ^infolist_former_and_current1
configuration1 :: (connectors1,L1,R1).



After instruction :
configuration2 : (connectors2,L1,R1).



Non-procedural description.

Before instruction : LRDCF leftsymbol_cf1
Params : in: ^infolist_former_and_current1

configuration1 :: (connectors1, L1, R1).
connectors1 :: (-, connector_current, -).
infolist_former_and_current1 :: (*, 0, -)+.
connector_current :: (-, local_projective_l1, -, active_r1, projective_r1).

After the instruction :
configuration2 :: (connectors2, L1, R1).
connectors2 :: connectors1 || connector_new.
connector_new :: (0, local_projective_l1, leftsymbol_cf1, ^infolist_former_and_current2,
0, projective_r1).
infolist_former_and_current2 :: infolist_former_and_current1:(*, leftsymbol_cf1, -)+.

Procedural description.

```
Procedure CF_REDUCE(leftsymbol_cf, infolist_former_and_current,
                    connector_current)
- denote leftsymbol_cf in all elements of infolist_former_and_current
- create a new connector with leftsymbol_cf, infolist_former_and_current and projective
                    of connector_current
- insert connector in connector queue
{ end of cf_reduce }
```

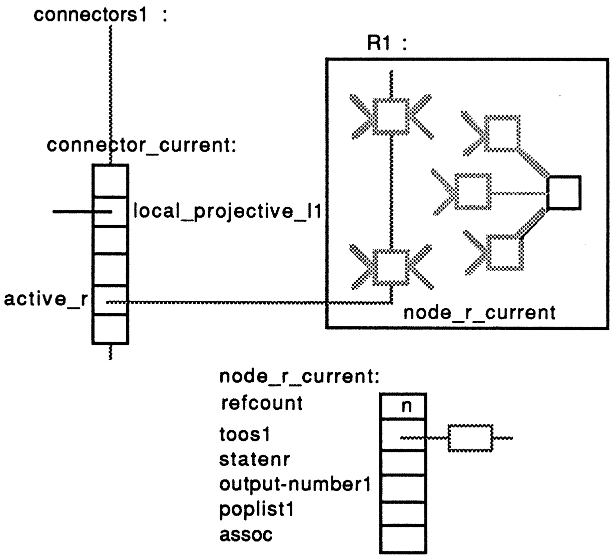
Instruction LRDCS

Informal description.

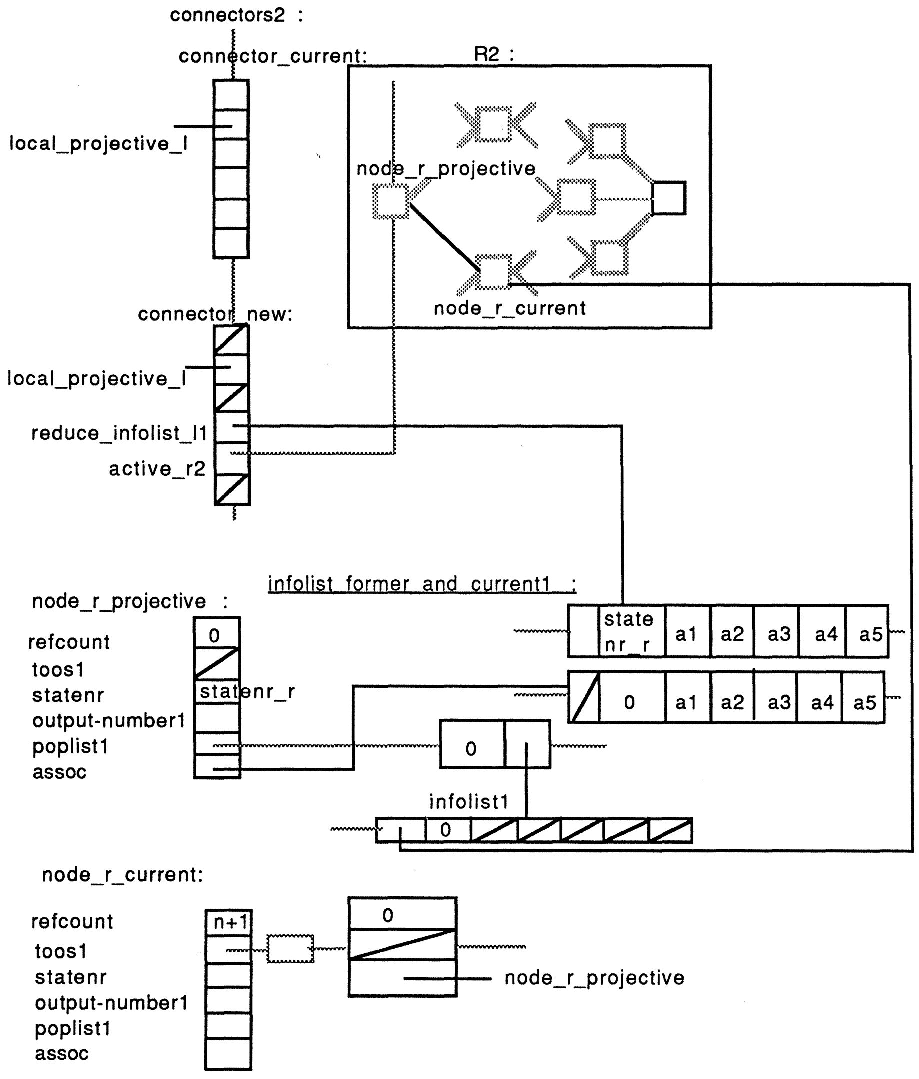
This instruction performs a cs reduce on the L-dag. It creates a new connector with infolist_former_and_current which provides for the returnstates and the associated parse trees. Moreover it contains a pointer to a new active_r which contains a newly created node in the R-dag which is connected as a gotostate to all the R-nodes in the current active_r.

Graphical description.

Before instruction : LRDCS statenr_r
Params1 : ^infolist_former_and_current1
configuration1 :: (connectors1,L1,R1).



After instruction :
configuration2 : (connectors2,L1,R2).



Non-procedural description.

Before instruction : LRDCS statenr_r
Params1 : ^infolist_former_and_current1

configuration1 :: (connectors1, L1, R1).
connectors1 :: (-, connector_current, -).
infolist_former_and_current1 :: (^D1, 0, -)+.
connector_current ::(*, local_projective_l, *, reduce_infolist_l1, -).

node_r_current1 :: (n, toos1, -).

after the instruction :

configuration2 :: (connectors2, L1, R2, Params1).
connectors2 :: connectors1, connector_new.
R2 :: R1, R_project.
connector_new:: (0, local_projective_l, 0, ^infolist_former_and_current2, active_r2, 0).
active_r2 :: ^R_project.
R_project :: (0, 0, statenr_r, 0, ^poplist1, ^infolist_former_and_current1).
poplist1 :: (0, ^infolist1).
infolist1 :: (^node_r_current, 0, 0, 0, 0, 0, 0).
infolist_former_and_current2 :: infolist_former_and_current1:(^D1, statenr_r, -)+.
infolist_former_and_current3 :: infolist_former_and_current1:(0, 0 , -)+.
node_r_current1 :: (n+1, toos2 , -).
toos2 :: toos1, (0, 0, R_project).

Procedural description.

```
Procedure CS_REDUCE (statenr_r, infolist_former_and_current,
                     connector_current )
- notestatenr_r in all elements of infolist_former_and_current
- create a new node_r and link it to all the node's of the active_r of connector_current
- create a new connector with active_r of connector_current and the rest as above
- insert connector in connector-chain
{ end of cs_reduce }
```

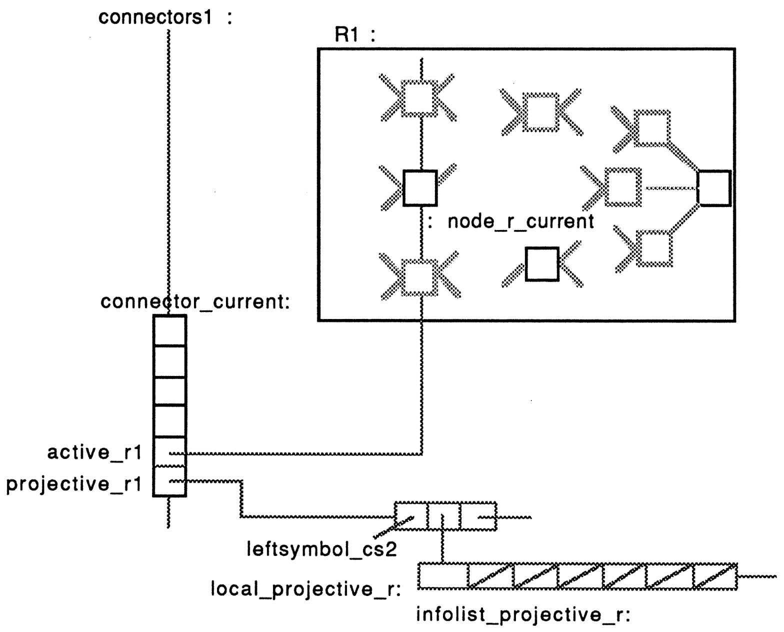
Instruction RGTS

Informal description.

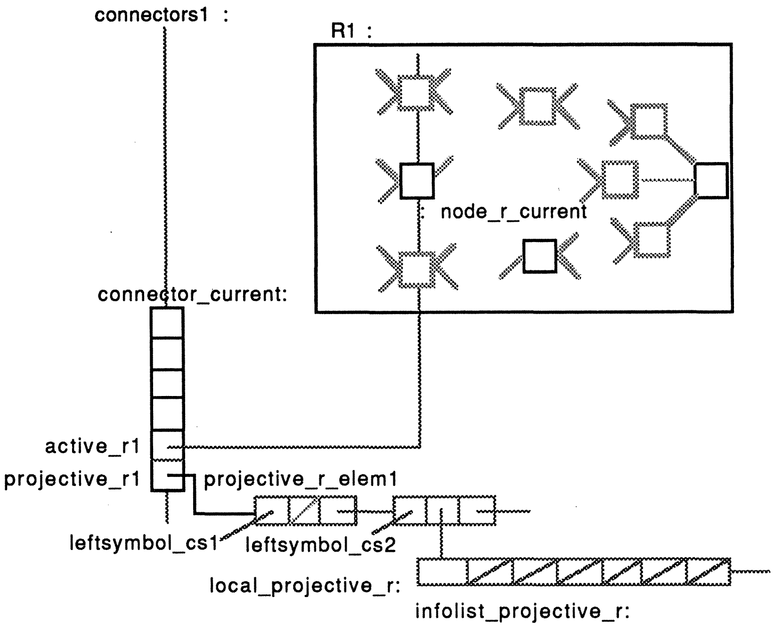
The instruction RGTS ("get a symbol from the R-dag") has as its argument the symbol on which an action can occur in the current R-node. That node is contained in the set active_r, which itself is contained in the current connector.
RGTS (or RGTSC) is the first instruction which is activated by the processor PR. It prepares an element of the projective_r of the current node_r by filling in the leftsymbol_cs which is provided as an argument of RGTS. The next instructions RSTK and/or RRDC will add to this element a (list of) projective and/or return-nodes which may serve as a new active_r.
We present the working of the instruction for the general case that more then one symbol can be generated for a R_state.

Graphical description.

Before instruction : RGTS leftsymbol_cs1
configuration1 : (connectors1,L1,R1,Params).
Params : out: projective_r_elem1



After instruction RGTS :
configuration2 : (connectors1,L1,R1,Params).



<u>Procedural description.</u>

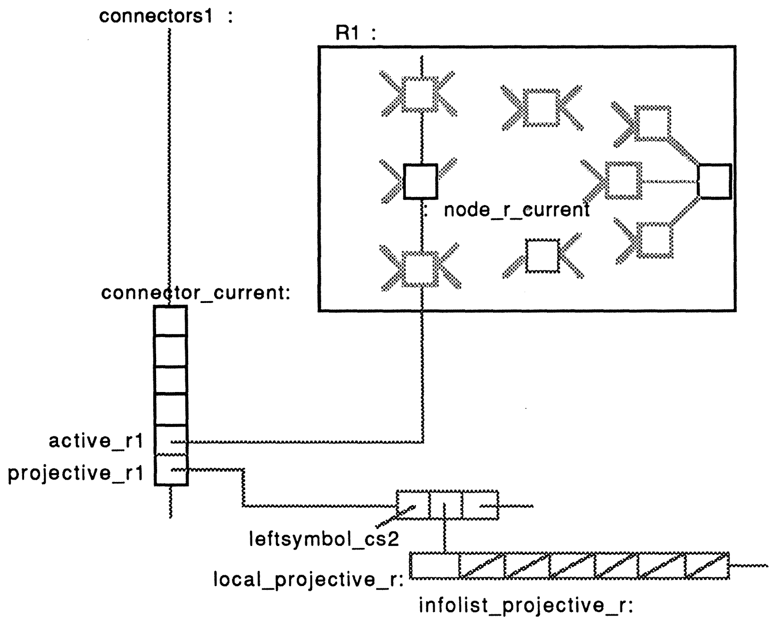
The procedural description of RGTS is contained within the processor PR.

Instruction RGTSC

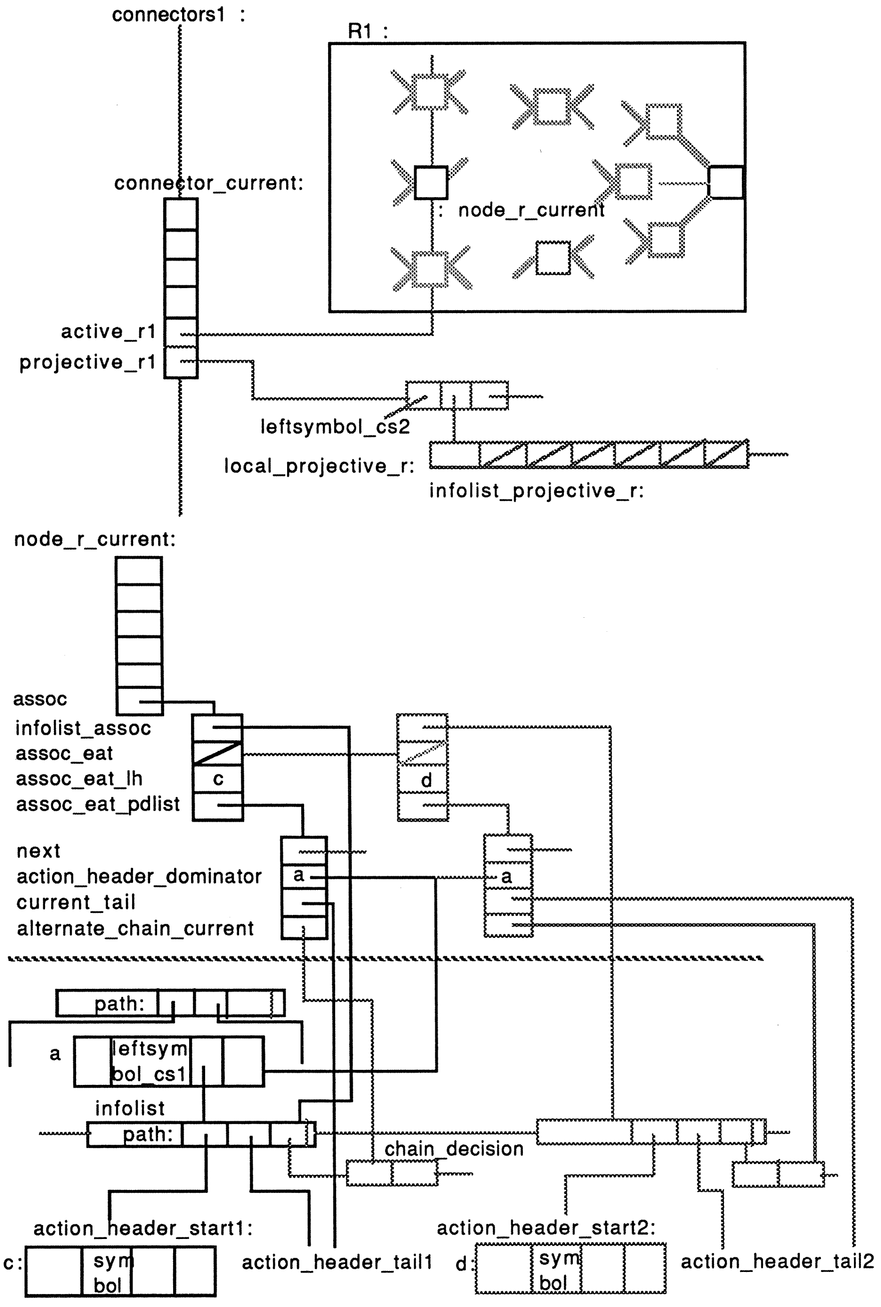
<u>Informal description.</u>
RGTSC (or RGTS) is the first instruction which is activated by the processor PR. It prepares an element of the projective_r of the current node_r by filling in the first leave of the cover_symbol leftsymbol_cs which is provided as an argument of RGTSC. The associated leaves are found in the association list of the R-node (an association list is a completed parse tree). If the statenr of node_r is 0 then it contains a non-nil pointer in an association list ("assoc-eat"). Input is then taken from that list. After all symbols are taken from the assoc then the normal processing of the code is resumed.
When the next symbol of an association list is taken a repositioning of the reading pointers assoc_eat and assoc_eat_lh is necessary. These pointers are moving up and down in the dag-structured parse forest. This repositioning is governed by the rules for pushing, shifting and popping in the forest.

Graphical description.

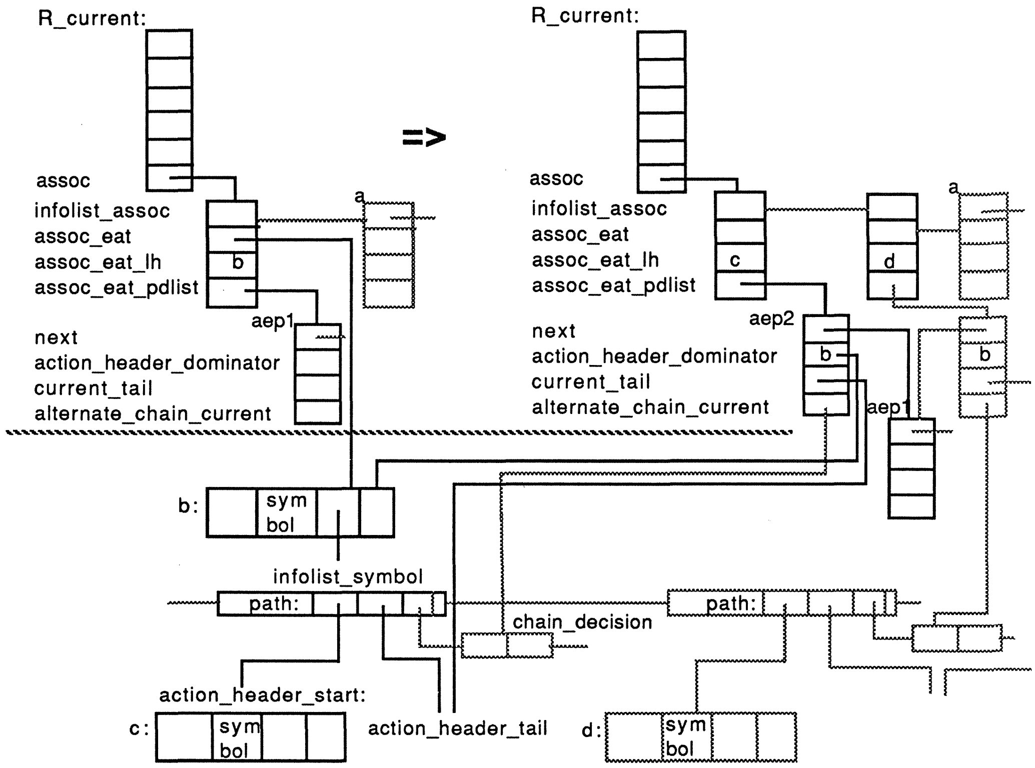
Before instruction : RGTSC leftsymbol_cs1
configuration1 : (connectors1,L1,R1,Params).
Params : out: projective_r_elem1



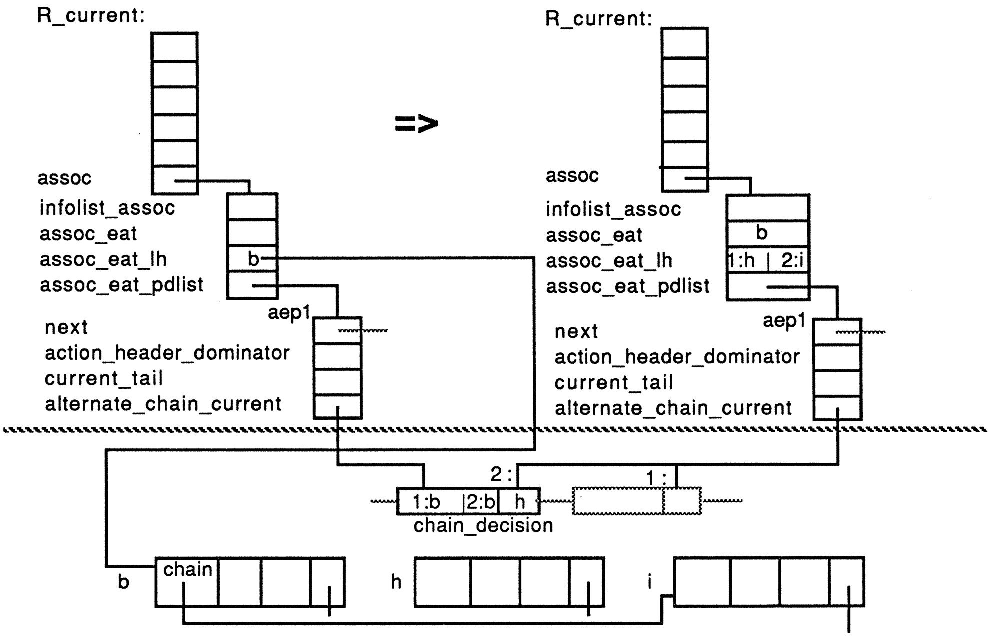
After instruction RGTSC leftsymbol_cs1
configuration2 : (connectors1, L1, R1)



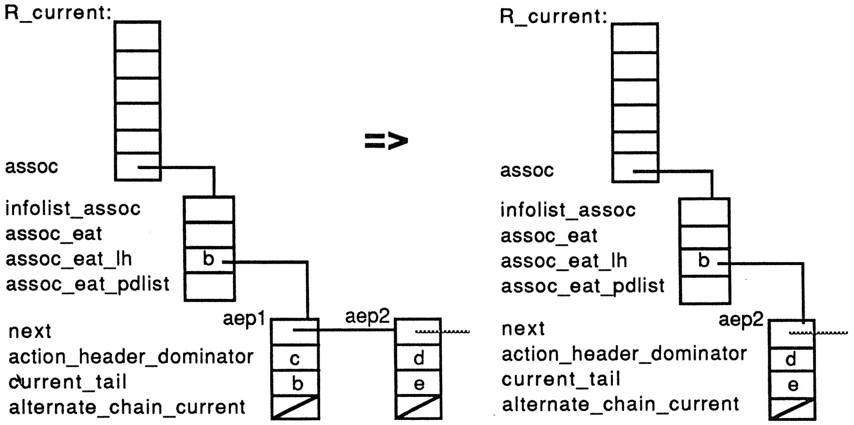
Push on nonterminal in assoc



Position on next terminal in assoc



Pop from end of current level in assoc



Procedural description.

The procedural description of RGTSC is contained within the processor PR. The procedural descriptions for the climbing within the parse forest are straightforward. They are omitted.

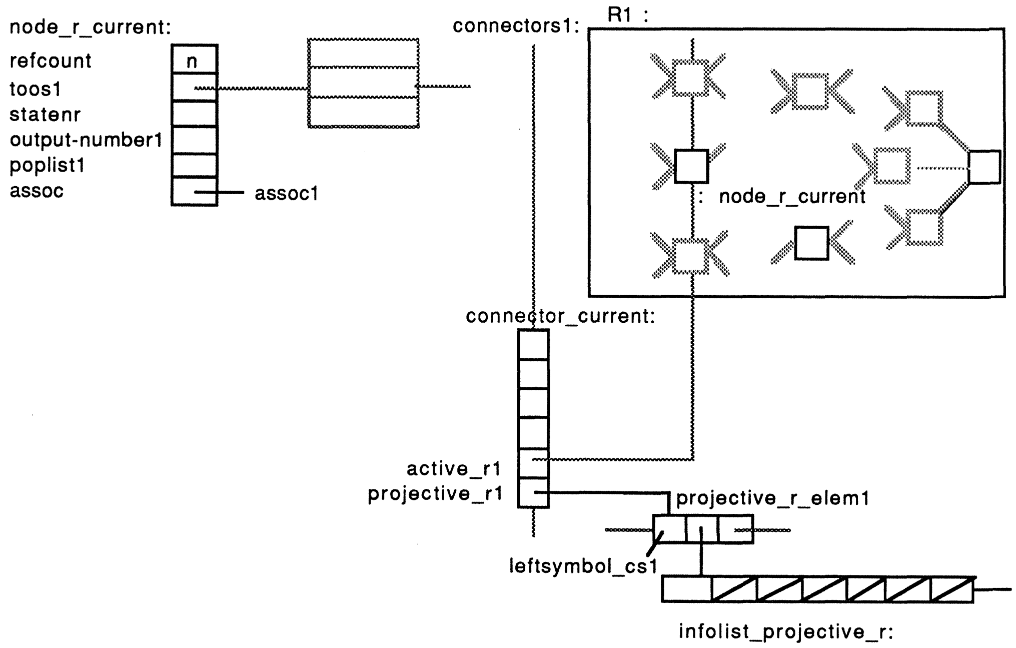
Instruction RSTK

Informal description.

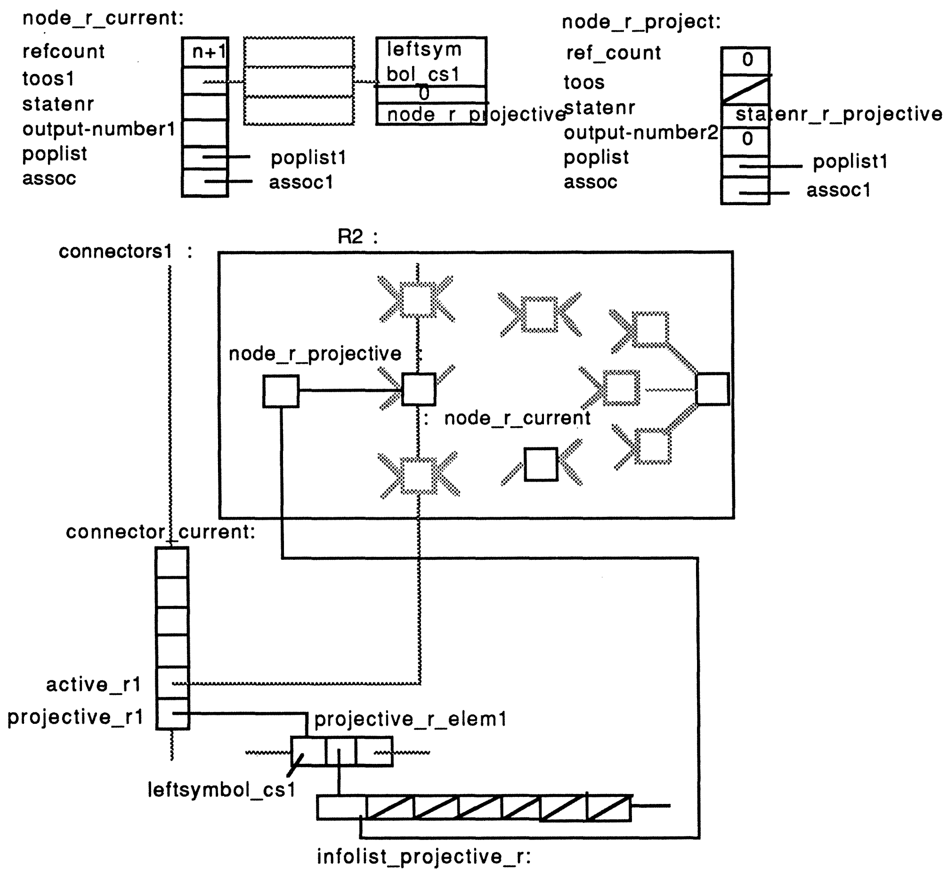
The instruction RSTK projects a node on the R-dag. An active_r and a local_projective_r both have the form of a reduce infolist. This in contrast with the L-dag, where the pointer to a projected L-node is stored in the projective nodelist, which contains only a fixed number of elements. With the instruction RSTK an otherwise empty infolist with a pointer to the projected R-node is added to the current projective_r_elem.

Graphical description.

Before instruction : RSTK statenr_r_proj
Params : in: ^node_r_current1, leftsymbol_cs1, ^projective_r_elem1.
configuration1 :: (connectors1,L1,R1).



After instruction : RSTK
configuration2 : (connectors2,L1,R2).



<u>Non-procedural description.</u>

Before instruction : RSTK statenr_r_proj
Params : in: ^node_r_current1, leftsymbol_cs1, ^projective_r_elem1.

configuration1 :: (connectors1, L1, R1).
connectors1 :: (-, connector_current, -).
connector_current :: (-, ^projective_r1).
projective_r1 :: (-, projective_r_elem1, -).
projective_r_elem1 :: (leftsymbol_cs1, local_projective_r1).
node_r_current1 :: (n, toos1, -, ^assoclist1).

After instruction:
configuration2 :: (connectors1, L1, R2).
R2 :: R1, R_project.

R_project :: (0, 0, statenr_r_proj, 0, ^poplist1, ^assoclist1).
poplist1 :: (0, ^infolist1).
infolist1 :: (^returnstate1, 0, 0, 0, 0, 0, 0).
projective_r_elem1 :: (leftsymbol_cs1, local_projective_r2).
local_projective_r2 :: local_projective_r1, (^R_project, 0, 0, 0, 0, 0, 0).
node_r_current1 :: (n+1, toos2 , - , ^assoclist1).
toos2 :: toos1, (^R_project, 0, 0).

Procedural description.

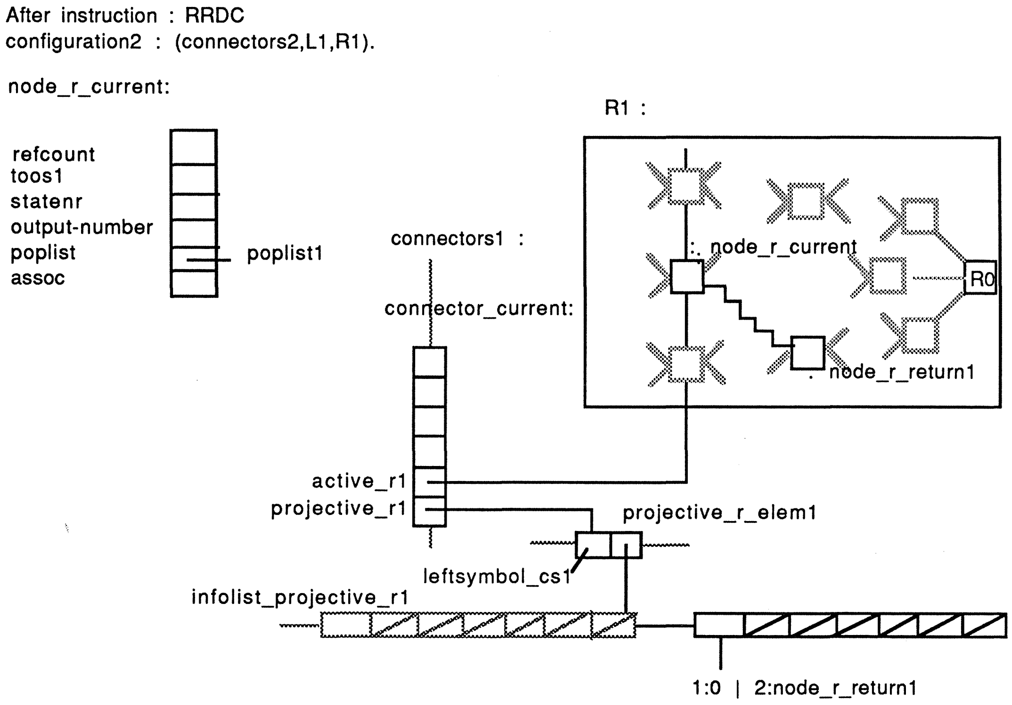
The procedural description of RSTK is contained within the processor PR.

Instruction RRDC

Informal description.

This instruction performs a reduce on the R-dag. The infolist of the reducing item (because of the current formalism there is only one item, indicated by a "*") is attached to the current projective_r_elem.

Graphical description.



Non-procedural description.

Before instruction : RRDC
Params: in: ^node_r_current1, ^projective_r_elem1.

configuration1 :: (connectors1, L1, R1).
projective_r_elem1 :: (leftsymbol_cs1, ^local_projective_r1).
node_r_current1 :: (n, -, ^poplist1, -).
poplist1 :: (*, ^infolist1).
infolist1 :: (^returnstate1, -) .

after the instruction ::

configuration2 :: (connectors1, L1, R1).
node_r_current1 :: (n-1, - , ^poplist1, -).
projective_r_elem1 :: (leftsymbol_cs1, local_projective_r2).
local_projective_r2 :: local_projective_r1, (returnstate1, 0, 0, 0, 0, 0, 0).

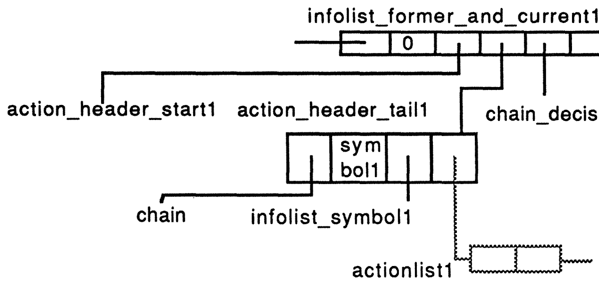
Instruction SRL

Informal description.

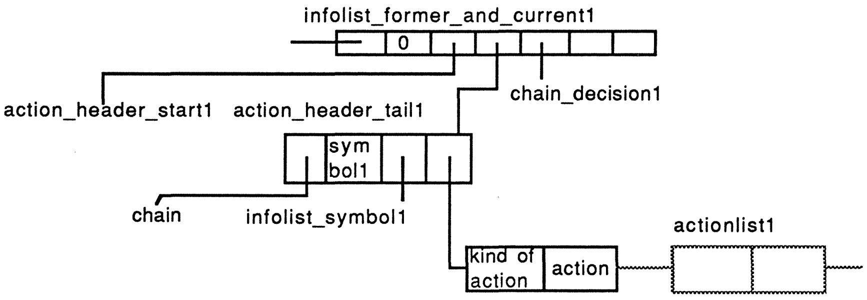
This instruction adds to the actionlist of the current actionheader of an infolist a reportnumber with an indicator of the kind of action, in this case a "report".

Graphical description.

Before instruction : SRL reportnr.
Params1 : in, out: ^infolist_former_and_current1.



After instruction : SRL reportnr



Non-procedural description.

Before instruction : SRL reportnr.

Params1 : in: ^infolist_former_and_current1.
infolist_former_and_current1 :: (*, *, *, action_header_tail1, -).
action_header_tail1 :: (*, *, *, actionlist1).

After instruction.

action_header_tail1 :: (*, *, *, actionlist2).
actionlist2 :: ('r', reportnr), actionlist1.

Procedural description.

The procedural description of SRL is contained within the processor PF.

Instruction ALL

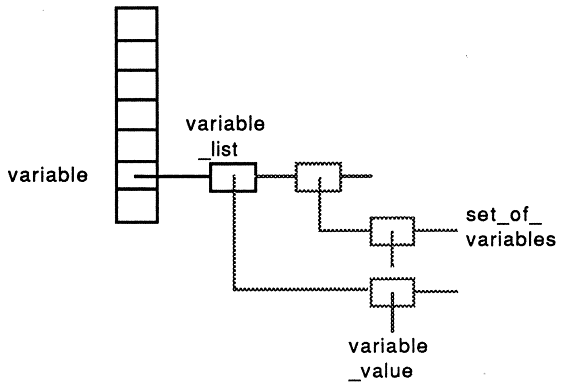
Informal description.

The instruction ALL creates room for the values of variables in the infolist of the current item.

Graphical description.

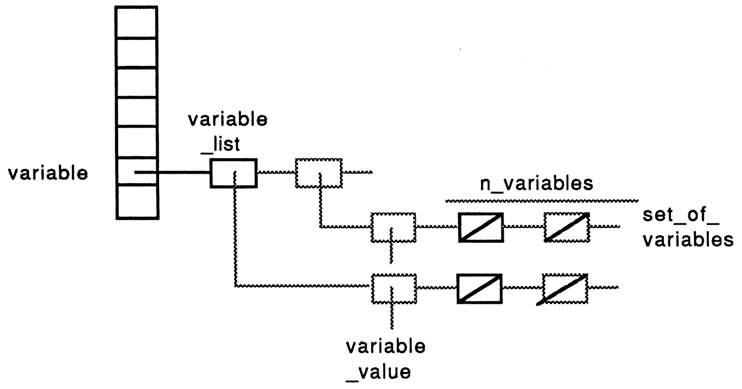
Before instruction: ALL n_variables
Params : in, out: ^infolist_former_and_current1.
configuration1 :: (connectors1,L1,R1).

infolist_former_and_current1 :



After instruction ALL :

infolist_former_and_current1 :



Non-procedural description.

Before instruction : ALL n_variables
Params: in, out : ^infolist_former_and_current1

infolist_former_and_current1 :: (-, variable1, -).
variable1 :: [variable_list1]*.

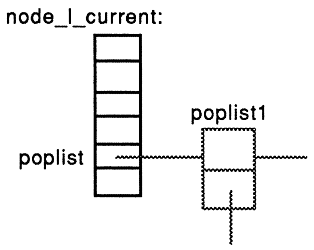
after the instruction :

variable2 :: [variable_list2]*.
variable_list2 :: variable1:(variable_list1, [(∅)]n_variables).

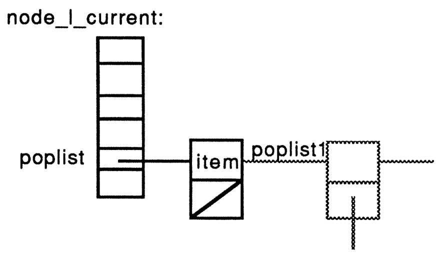
Instruction CLI

Informal description.
Normally, a starting item is not represented in a runstate. With the instruction CLI it is created. It has to be present in order to receive the actual values of variables which are denoted as formal parameters with the calling nonterminal.

Graphical description.
Before instruction: CLI item
Params : in, out: ^node_l_current



After instruction :CLI item



Non-procedural description.

Before instruction : CLI item
Params: in, out: ^node_1_current

node_1_current :: (-, poplist1, -).

after the instruction :

node_1_current :: (-, poplist2, -).
poplist2 :: (item, ∅), poplist1.

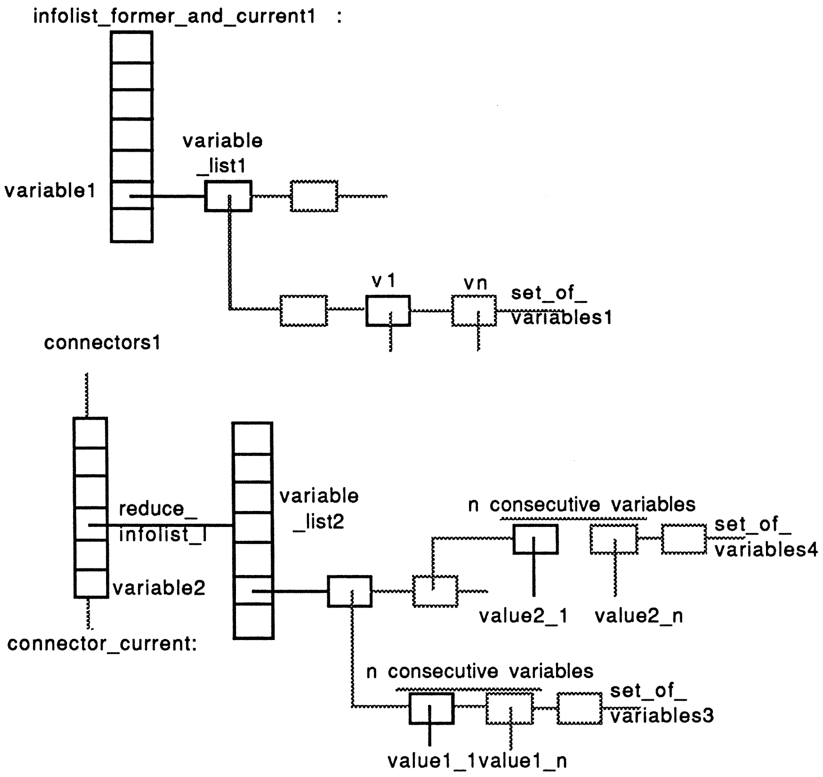
Instruction RCV

Informal description.
The instruction RCV is generated when a nonterminal symbol for which formal parameters were denoted acts as an input symbol. The nonterminal symbol stems from a connector which contains also a reference to its related infolist_symbol. Within that infolist_symbol are stored the variables of the reduced nonterminal, together with their values. These values have

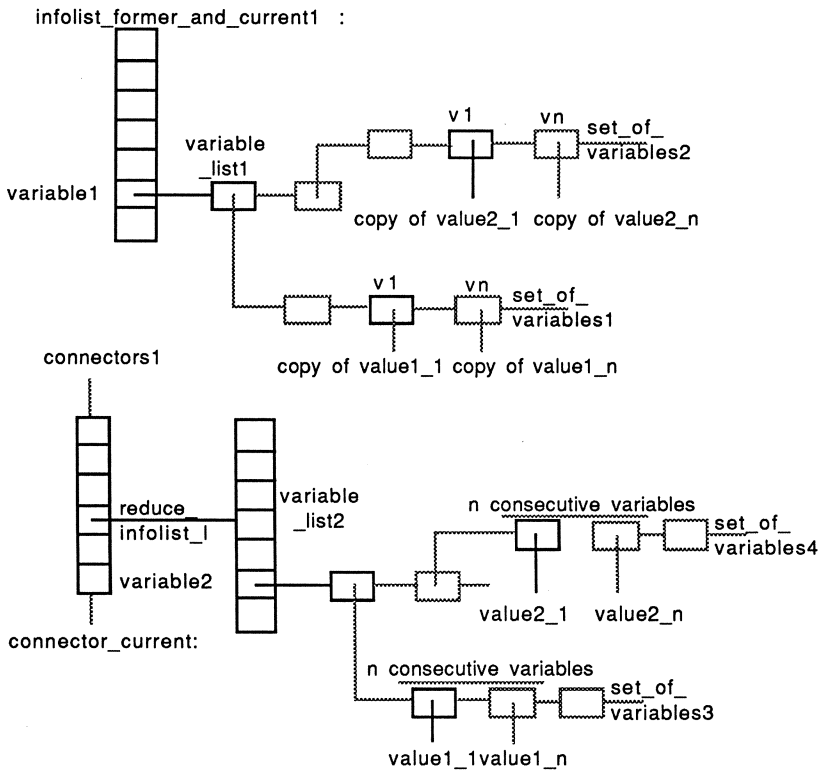
to be transported to the infolist_former_and_current of the current item. Within infolist_symbol the values of the variables are read sequentially.

Graphical description.

Before instruction: RCVn v1...vn
Params : in: ^symbol_infolist, ^infolist_former_and_current1
out: ^infolist_former_and_current1
configuration1 :: (connectors1,L1,R1).



After instruction :RCV n v1...vn
configuration1 :: (connectors1,L1,R1).



Non-procedural description.

Before instruction : RCV n v1...vn
Params: in: ^symbol_infolist
in, out: ^infolist_former_and_current1

configuration1 : (connectors1, L1, R1).
connectors1 :: (-, connector_current, -).
connector_current :: (-, reduce_infolist_l, -).
reduce_infolist_l :: (-, variable2, -).
variable2 :: [variable_list2]+.
variable_list2 :: parameter_variables, local_variables1.
parameter_variables :: [value_i]n
local_variables1 :: [variables]*.
infolist_former_and_current1 :: (-, variable1, -).
variable1 :: [variable_list1]*.
variable_list1 :: [variables]*.

after the instruction :
configuration2 : (connectors1, L1, R1).
for i=1..n : $\text{variable\_list1}_{v(i)}$.variable_value = $\text{parameter\_variables}_n$.variable_value.[3]

Procedural description.

The procedural description of RCV is straightforward and is not presented here.

Instruction SND

Informal description.
The instruction SND is the counterpart of the instruction RCV. It copies values of variables from the current infolist_former_and_current to the infolist of a closure item in node_l_projective. That closure item was just before created by the instruction CLI. The process of copying is the same as described with RCV and will not be duplicated here.
For a better understanding of the connection of the different instructions we refer to the section on variables in chapter 6 where code generation is discussed.

The instructions PSH, CAT, ASS, TEQ, TNE

Informal description.
For a description of the instructions PSH, CAT, ASS, TEQ and TNE we refer to the overview in section 4.2.2.5. They operate on a separate linear stack which serves solely for the evaluation of an expression with variables. The variables themselves are found in the infolist_former_and_current which is prepared by the preceding instruction TOP. Implementation of the instructions is straightforward and is not further discussed here.

Instruction LEXST

Informal description.
The instruction LEXST initialises the lexicon_pointer "place" which is located in the field "lexicon" of infolist_former_and_current. Its accompanying field, "is_an_entry" will get the value "false". The initial pointer is set on the start of a trie structured lexicon, as described in chapter 2. The handling of the pointer is done by external procedures which are described in (Skolnik, 1982).

Instruction LEXINC

Informal description.
The instruction LEXINC increments the pointer "place" in the lexicon along the current input symbol. Before the execution of the instruction the field "is_an_entry" will be initialised on "false". It will get the value "true" when the pointer will be positioned on the end of a lexicon_entry.

[3] This is one of the places where the unifying formalism falls short in expressing power for a typical programming purpose : the indexing in an array or list.

Instruction LEXRDC

Informal description.
The instruction LEXRDC performs a reduction just like the instruction for a cf reduction (LRDCF) when the field "is_an_entry" within infolist_former_and_current is "true". The reduction is made by the creation of a connector which contains infolist_former_and_current. After the entry in the lexicon one or more categories may be present. The first category is placed in infolist_former_and_current.nonterminal. The other categories are transferred as values of variables, in the sequence in which they are present, into infolist_former_and_current.variable. Multiple values of a category will be stored as multiple values of a variable.
We refer to chapter 6 for a description of the way in which code is generated in order to continue in the lexicon after the current entry. When no continuation is possibly the infolist_former_and_current will be pruned automatically.

4.2.2.6 PP : processor for pruning

4.2.2.6.1 Pruning of connectors

If a connector becomes instable because no further processing is possible then its reference count becomes 0. The general pruning mechanism will then be put into operation.

4.2.2.6.2 Pruning in general

If the reference count of a record becomes 0, then the reference count of all records which were referenced by this record will be decreased by 1.

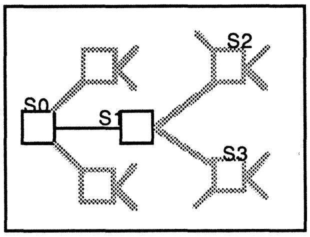
4.2.2.6.3 Transduction

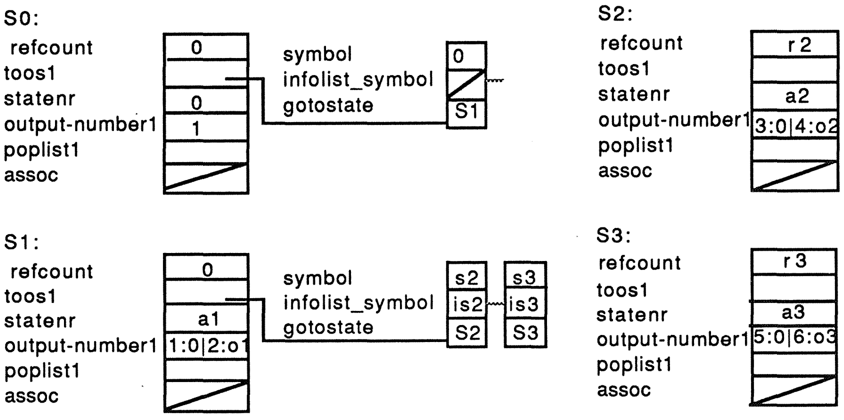
The mechanism of the reference count is also used for the creation of output with finite delay (see chapter 2). The following condition is checked permanently.

Online transduction.

Before :

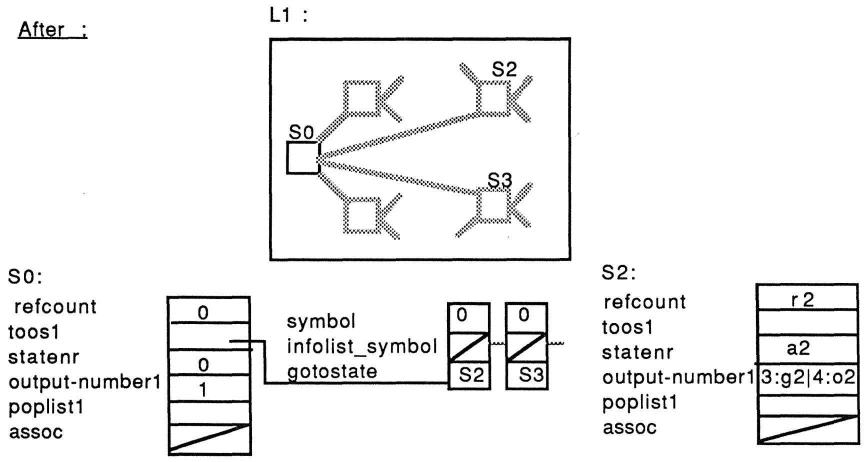
L1 :

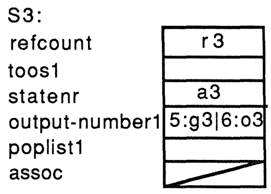




transductionfile = T

N.B.: "g" is abbreviation of "genint" : a unique generated integer





transductionfile T = T, ([1:g1|2:o1], [3:g2|4:o2],s2),
([1:g1|2:o1], [5:g3|6:o3],s3).

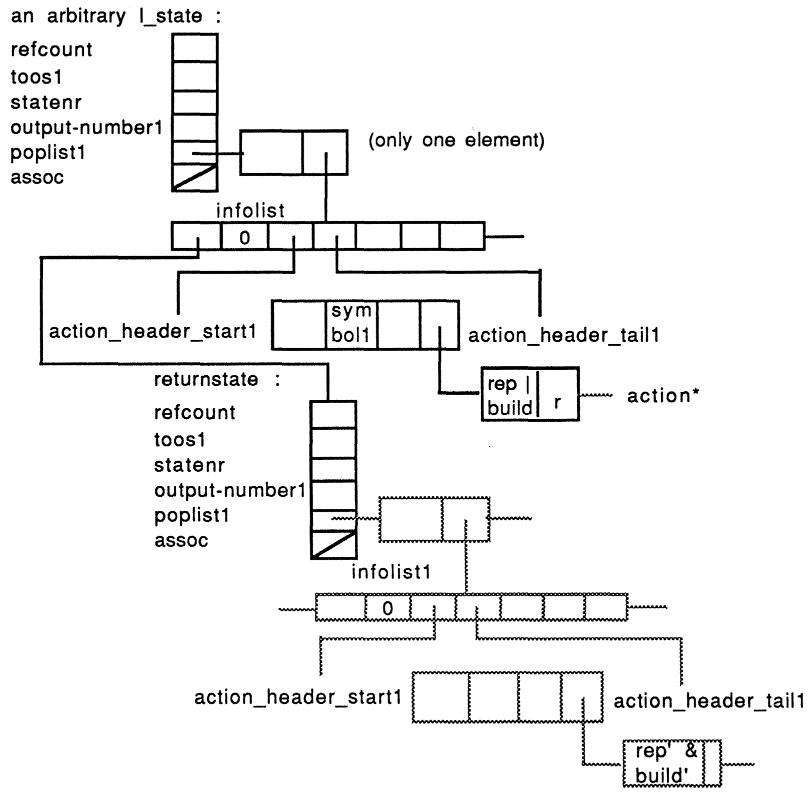
n.b.: if s is nonterminal : the leaves of its infolist_symbol are brought to the transducefile instead of s itself.

4.2.2.6.4 Reporting and structure building

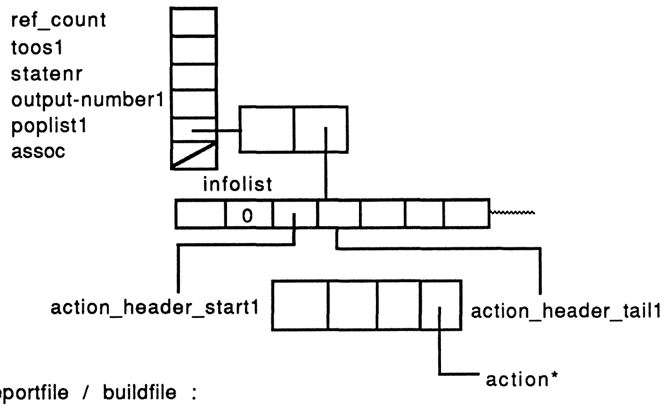
The same mechanism that handles output functions also for reporting and structure building.

Online reporting and structure building :

Before :



After :



reportfile / buildfile :
r symbol1

4.3 Generation with a PTA

The PTA can be extended by a minor addition in order to produce sentences instead of reading them. The processor PT reads the input which is then treated by the L-nodes in the active nodeset. For each of these L-nodes the expected terminal symbols may be sampled. If these expected symbols are merged for the whole active nodeset a message can be generated, before the reading of a symbol, which symbols are admissible. Instead of reading a character one character of this set may be selected as the next input character.

4.4 Undecidability

It can not be guaranteed that the processors will terminate. The recognition of type-0 languages is in principle undecidable. The processors PL and PR will always terminate (as does LR-, Earley- and Tomita-parsing), but during processing they may create new connectors which will activate PL and PR again. For instance, in the case of a type-0 grammar with only lengthening rules the recognition will not terminate. Another reason for non-termination is caused by a circularity between lhs's and rhs's of grammar rules.
However, some detection of potential non-termination can be build in. A connector contains all information which is necessary to activate the PL and PR processors. After the creation of a new connector it can be determined if there is another connector C with the same information. It is not important whether the processing of C may have been in the past, may occur in the present or will happen in the future. It is only important that a piece of unique work will be done only once. If a new connector already exists there is no need to keep the new connector.

4.5 Parallel processing

In chapter 7 we will consider the complexity of the PTA. In chapter 2 we suggested that further improvements in speed of processing can be obtained by making use of special hardware. In this section we will investigate how the PTA can be optimized by making use of parallel processing.
In the literature only a few papers appeared on parallel parsing and compiling. Cohen and Kolodner (1985) presented a short overview from which we borrow some descriptions.
Fischer (1975) divides the input strings into segments, one for each processor. Several stacks may have to be kept by each processor because a processor does not know the state in which its left neighbour will be when that neighbour finishes scanning its segment. The grammar of the language being parsed, however, is deterministic. Fischer's algorithm is synchronous. This means that at each point in the computation, each processor tries to perform the same operation. Only after all processors have finished this operation they proceed to the next one. The results of simulating the model on a computer indicate that substantial gains in speed are possible when using several processors.
Mickunas and Schell (1978) show how to design scanners (including table lookup) which operate in parallel. Further they extend the LR parsing technique to the case of multiple processors which can start parsing at an arbitrary place in the input string.
Lipkie (1979) considers the compilation of Algol-like programs using multiple independent processors. He deals with two kinds of concurrency: one in which processors perform simultaneous separate compilations of procedures of comparable size, the other in which several processors simultaneously execute the various multiple passes of a compilation (e.g. scanning, parsing, code generation, etc.) The results of his simulations indicate that the speedup of compilation varies linearly with the number of available processors.

Schell (1979) proposes a parallel variant of LR parsers having two additional operations *cancel* and *continue* besides the usual *shift* and *reduce*. These operations are needed for performing the merging operation of different stacks.
Cohen and Kolodner (1985) themselves describe research done in the line of Fischer.
Fisher (1985) describes an implementation of an LL(1) based algorithm for the parsing of Van Wijngaarden Grammars. Each hyperrule in the grammar is delegated to a separate processor. He makes use of the concurrent programming language Occam.
Rytter (1987) describes an algorithm for the recognition of unambiguous cf languages in O(ln(n)) time on a polynomial number of parallel processors.

Summarizing we may conclude that all this research is based on deterministic parsing, attempting to accelerate by
- sending pieces of the input text to different processors
- pipelining the different phases of syntactic processing
- trying different alternatives in a grammar by separate processors.

Acceleration of Earley's algorithm by parallel hardware is reported by Chang and Fu (1984). They use a triangular shaped VLSI array. This array system has an efficient way of moving data to the right place in the right time. Simulation results show that this system can recognize a string with length n in 2n+1 system time.

We are concerned with the parsing of ambiguous and enriched type-0 grammars. The recognition/parsing/transducing is performed by our PTA. Together with the compilation process (which is able to translate conceptually parallel transduction rules into a sequential transducer) in a number of cases a polynomial runtime performance is guaranteed. Our approach will be to improve the runtime by exploiting the possibilities within the PTA for parallel processing. In the cases that the PTA behaves like a PDA or a FSA the above mentioned accelerations may also be considered.
The "P" in "PTA" now gets a double meaning : Parallel for parallel rules in the formalism and Parallel for parallel processing. But there is not necessarily a direct relation between the two.

Proposal for the implementation of the PTA on parallel hardware based on transputers.

In general, when a number of hardware processors are operating in parallel, one processor may communicate with a number of other processors. Some or all processors may share a piece of common memory, and some of them will have access to local memory. Therefore, a number of different formal machines could be devised to cope with different possible hardware configurations.
We will consider here a hardware configuration which recently attracted much attention because of its reasonable price and high speed of processing : the transputer. The Occam language mentioned above is able to govern a number of these transputers. In a multiprocessor system based upon transputers a number of processors (with a small number of simple instructions), each with their own local memory, interact with other processors. The configuration is either fixed or can be reorganised by a special processor which creates the links between some processors. There is no common memory .
From a software point of view each processor may be seen as an actor on a piece of memory. To map the PTA on a system of transputers we could assign to each processor the task of managing the operations for a specific datastructure. As such we may speak of "object-oriented processing". In case of frequent access the "intelligent memory" for one datastructure may be "divided" into several "pieces": more processors which perform the same kind

of tasks, but each with their own piece of memory. In that case one processor may act as a supervisor for the management of a specific datastructure.
Because there is no common memory no conflicts will arise during the updating of a datastructure: they are resolved automatically by the hardware of the connection between the processors.

In the PTA inherent parallelism is indicated by the appearance of regular expressions in the grammar descriptions. As such they are found in :
- the set of all connectors (to be found in "configuration")
- within a connector : the set of all active nodesets (to be found in "nodeset")
- for each runstate : the set of all applicable input-symbols (to be found in the connected nodesets of R and in parallel input)
- for each runstate : the set of all applicable instructions TOP (to be found in processor PL : [list]*)
- for each runstate : the set of all applicable instructions SRL (to be found in processor PL : [action]*).

Instructions are pointed to by a program counter. Each parallel process will have its own program counter and stack of program counters (for nested subroutines of instructions). We assume the availability of a translation table for compile-statenumbers (in QL or QR) into a starting program counter.

Following this schema of thought little has to be changed in the definition of the PTA. The potential applications are numerous : if a problem can be specified with the aid of the unifying formalism of chapter 2 it will be possible to process it on a parallel computer without any reformulation of the problem.

5. Extension of the LR-table construction method for the unifying formalism

In this chapter we will show how the algorithms for the construction of LR(0) itemsets can be extended for the treatment of the various extensions of the unifying formalism.

In section 2.6 we formulated as our goal the development of a program generator for
- type-0 and transduction grammars
- grammars with regular expressions
- parallel parsing
- grammars which contain don't cares, arb's, line's, ranges of symbols and tree symbols
- grammars which contain variables
- grammars which contain boolean expressions
- grammars which contain report- and build-functions.

In the chapters 3 and 4 we discussed how to handle type-0 and transduction grammars with the PTA, assuming that code can be generated for lhs's and rhs's separately. The PTA handles also the parallel parsing. The remaining extensions are handled by the compiler.
In this chapter we will discuss the essential ideas for the extension of the LR(0) algorithm in order to handle the other sub-formalisms. The ideas will be expressed within the original formalism of a U-grammar. In the next chapter the implementation of the whole algorithm is discussed, together with a number of preprocessing algorithms.

In this chapter we will discuss the extension of the algorithm for the creation of LR-tables for the following sub-formalisms :
- regular expressions
- concerning symbols :
    - ranges of terminal symbols
    - tree symbols
    - "don't cares"
    - "arb's" and "lines"
- concerning Boolean expressions :
    - cooperation of grammar rules and the Boolean "and"
    - negations

5.0 Recapitulation of the LR(0) parser-generation algorithm

We will recapitulate the construction of LR-tables according to (Aho, Sethi and Ullman, 1986 pp. 222-232) in a slightly different way in order to adjust to the algorithms which will be presented in subsequent chapters. The differences are :
1. in an LR-table no distinction is made between terminals and nonterminals (there is only an "action"-part);

2. no itemset will contain items of the form "A::α."; on the appearance of an item of the form "A::β.X" a reduce instruction will be generated on an X;
3. an LR-table may contain multiple entries.

Definition.
An *LR(0) item* is of the form

$[A::\alpha.\beta]$,

where $A::\alpha\beta$ is a production. Alternatively, we sometimes denote it as a 2-tuple

$[p, j]$,

where p is a production $A::\alpha\beta$ and j is an integer representing the position of the dot ($j=|\alpha|$).
Δ

Definition.
CLOSURE(s) is a function that takes a set of items s as its argument and returns another set of items. Calculation of CLOSURE(s):

```
- repeat
   - for each item [A::α.Bβ] in s and each production B :: γ such that [B::.γ]
     is not in s do
       - add [B::.γ] to s
- until no more items can be added to s
- return s.
```

Δ
We will call the item $[A::\alpha.B\beta]$ the "father item" of the "closure item" $[B::.\gamma]$.

Definition.
GOTO-ITEMSET(I, X) is a function that takes an itemset I and a grammar symbol X as its arguments, and returns another itemset.
Calculation of GOTO-ITEMSET(I, X) :

```
- Let I' be the set of items [A::αX.β], such that [A::α.Xβ] is in I.
- return CLOSURE(I').
```

Δ

Definition .
CONSTRUCT-ITEMSETS is a function that constructs a set of itemsets, C. A special production, S' :: (, S,). , is introduced.
Calculation of CONSTRUCT-ITEMSETS:

```
- let C be {{[S' :: .(, S, )]}}
- repeat
   - for each itemset I in C and each grammar symbol X such that GOTO(I, X) is
     not empty and is not already in C do
       - add GOTO(I, X) to C
- until no more sets of items can be added to C.
```

Δ

Definition.
LR-TABLE(G) is a function that takes a grammar G augmented by the production
S' :: (, S,). as its argument, and constructs an ACTION table.

Calculation of LR-TABLE(G) :

```
- Construct $C = \{I_0 ... I_n\}$ by CONSTRUCT-ITEMSETS
- for i := 0 to n do
  - if $[A :: \alpha.X\beta]$ is in $I_i$ and GOTO($I_i$, X) = $I_j$
    - add 'shift j' to ACTION(i, X)
  - if $[A :: \alpha.\ X]$ is in $I_i$
    - add 'reduce p' to ACTION(i, a), where p is a production $A :: \alpha$
  - if [S' :: (, S, .)] is in $I_i$
    - add 'accept' to ACTION(i, ")").
```

The initial state of the parser is the one constructed from the set containing item [S' :: .(, S,)].
Δ

We will discuss in the following subsections for each of the sub-formalisms separately how they can be handled by a modification of the function GOTO-ITEMSET and, eventually, the function CLOSURE. Thereafter we will treat the general case of the combination of all the extensions. The extension of the procedures CONSTRUCT-ITEMSETS and LR-TABLE are dealt with in the next chapter.

In the definition of GOTO-ITEMSET we will call I' the "core" of the created itemset. The returned itemset is a "successor itemset" of I. The definition of I' can be reformulated as :

if $[A::\alpha.X\beta]$ is in I and X is the grammar symbol appearing in the call to GOTO-ITEMSET then $[A::\alpha X.\beta]$ is in I'.

In the sequel we will abbreviate this definition by the following notation :

(a) $.X\beta$ --X---> $X.\beta$

In this notation the contribution of an individual item is highlighted. We will say that an item "travels" along a symbol (in this case X). The traveling will result in the LR-table in a shift-action on X if $|\beta| > 0$ and in a reduce-action on X if $|\beta| = 0$.

The extensions to the function GOTO-ITEMSET will be formulated in this notation.

In our notation we will allow strings α, β, .. (in V*) to contain regular expressions, as we defined them in the unifying formalism. Symbols X (in V) and the opening square bracket of a regular expression may bear the negation marker.

In the notation (a) we assume that the dot in the left side is placed before an X. In the right side the dot is placed before the β which may start with a square bracket of a regular expression. We will consider the position of the dot before an X or an X' as being stable, else the position is considered to be unstable. An item which contains a dot in an unstable position will be called an unstable item. In order to reach in an automatic way a stable position we introduce transformation rules for unstable items. They are of the form

(b) $.[\alpha]+\beta$ => $[.\alpha]+\beta$

It is possible that an unstable item goes through a series of transformations before it becomes a stable item. During the transformation(s) it may create other items, which may be either unstable or stable.

The right side of a transformation may be empty, for instance in :

(c) α.`]` => ε

(the notation .` will be introduced later on). In that case there is no continuation for the unstable item.

The transformations of an item belong to its traveling. The resulting stable item(s) will determine if the traveling results in a shift- and/or one or more reduce-actions in the LR-table.
We will use the notation $A :: .\alpha \mid .\beta$ as a shorthand for the writing of the 2 items $A :: .\alpha \mid \beta$ and $A :: \alpha \mid .\beta$.

5.1 Regular expressions

Informal.
Regular expressions were defined in section 2.2.2 . It is straightforward to write down for each of these definitions the necessary transformations.

Formal.

$$\begin{array}{lcl}
.[\alpha \mid \beta \mid ..]\ \gamma & => & [.\alpha \mid .\beta \mid ..]\ .\gamma \\
.[\alpha \mid \beta \mid ..]1\ \gamma & => & [.\alpha \mid .\beta \mid ..]1\ \gamma \\
.[\alpha \mid \beta \mid ..]*\ \gamma & => & [.\alpha \mid .\beta \mid ..]*\ .\gamma \\
.[\alpha \mid \beta \mid ..]+\ \gamma & => & [.\alpha \mid .\beta \mid ..]+\ \gamma \\
 & & \\
[\alpha \mid \beta. \mid ..]\ \gamma & => & [\alpha \mid \beta \mid ..]\ .\gamma \\
[\alpha \mid \beta. \mid ..]1\ \gamma & => & [\alpha \mid \beta \mid ..]1\ .\gamma \\
[\alpha \mid \beta. \mid ..]*\ \gamma & => & [.\alpha \mid .\beta \mid ..]*\ .\gamma \\
[\alpha \mid \beta. \mid ..]+\ \gamma & => & [.\alpha \mid .\beta \mid ..]+\ .\gamma
\end{array}$$

5.2 Symbols

In the definition of GOTO-ITEMSET the symbol X may be a nonterminal, an intermediate symbol or a terminal. In chapter 2 we defined for a U-grammar that the set of terminals T need not be finite. We allowed a terminal to be a range of characters. We therefore introduce for an itemset I the symbol "." as the "rest symbol". It is defined as the set $T-\{a \mid a \in T$ and I contains an LR-action on a$\}$. It will always be denoted in the context of the itemset for which it is defined.
In chapter 4 we defined for a PTA also that the set of terminals T need not be finite.We therefore introduce for an itemset I the symbol "#" as the "don't care input symbol". It represents the set T itself.

5.2.1 "Don't cares"

Informal.
Suppose we are in the position to construct the successors of the itemset I_i which contains, among others, the 2 items :
(1) $A :: \alpha\ .\ a\ \beta$
(2) $B :: \beta\ .\ *\ \gamma$, where $\alpha, \beta, \gamma \in V^*$. On the shift on an "a" we construct a new itemset I_j which contains, at least, the item

$A :: \alpha\ a\ .\ \beta$

What about item (2) ? The "don't care" * represents any symbol $\in$ T, inclusive the "a". The conclusion is that this item may also travel along the symbol "a" to I_j where it becomes

$B :: \beta\ *\ .\ \gamma$

It is clear that item (2) may travel along any symbol ∈ T, inclusive the rest symbol ".". On the "." we therefore create a successor itemset with the shifted second item (together with all the items where the dot was present before a "don't care").
The same technique may be followed for "don't cares" in the text. During construction of the successor itemset for "#" all items may "travel" along this character.

Formal.

. * β --a---> * . β for all a ∈ (T+I)

5.2.2 "Arb's"

Informal.
Suppose we are in the position to construct the successors of the itemset I_i which contains, among others, the item :

A :: α . = a β

The "=" means in this case : the longest sequence of symbols which are not "a". Alternatively, we may write it as [a`]*. Literally speaking, the 'a' forces the item to shift over the 'a'. All other characters will leave the item intact.

Formal.

```
. = a β        --a--->        = a . β
               --a`--->       . = a β
```

5.2.3 "Lines"

Informal.
Suppose we are in the position to construct the successors of the itemset I_i which contains, among others, the item :

A :: α . - a β

The "-" means : any sequence of symbols. Alternatively, we may write it as [*]*. Literally speaking, the 'a' splits the item: one shifts over the 'a', the other one keeps itself intact. All other characters will leave the item intact.

Formal.

```
.-             =>             .-.     for all dot types
.-             --.--->        .-      for all dot types
{These two rules imply the following rules :
. - a β        --a--->        . - a . β
. - a β        --a`--->       . - a β
}
```

Example in combination with don't cares :

Suppose we have the pattern grammar

S :: - a a
S :: - * b

The starting itemset contains the two items :

S :: . - a a

S :: . - * b

We then construct, as the successor itemset for "a" :

S :: - a . a
S :: - * . b
S :: . - a a
S :: . - * b

and as the successor itemset for the rest symbol "." :

S :: - * . b
S :: . - a a
S :: . - * b

5.2.4 Ranges of terminal-symbols

Informal.
During the construction of an itemset it is possible that a number of items contain a dot before a range of terminal symbols. For instance the itemset contains :

S :: . a..g β_1
S :: . c..p β_2
S :: . e β_3

In that case we determine non-overlapping sub-ranges, in this case :

a..b, c..d, e..e, f..g, h..p and the rest symbol "."

The sub-ranges can be determined before the construction of successor itemsets. They are then treated as normal symbols. For each of them the itemsets have to be constructed.

Formal.
Not necessary. In the implementation a new set of action symbols must be calculated for each itemset.

Example.
In the example above the successor itemset on "c..d" will contain :

S :: a..g . β_1
S :: c..p . β_2

The successor itemset on "e..e" will contain :

S :: a..g . β_1
S :: c..p . β_2
S :: e . β_3

5.2.5 Tree symbols

Informal.
The tree symbols ':(', '(' and ')' are treated as normal terminals, with two exceptions. The first one concerns the applicability of a 'line'. We defined in chapter 2 that a line matches 0 or more arbitrary characters, except the ')'. With other words : the line operates only on the current level in the input-tree. We will therefore make a restriction on the rules which were defined in section 5.2.3 . The second exception concerns the skipping in the input of a labeled sub-tree.
'(..(' and ')..)' have to be written as pairs.

'(..(' can be rewritten as [* :(-]^n, ')..)' as [-)]^n , with n >= 0. When '(..(' and ')..)' are treated as nonterminals then the treatment of the n can be written in the unifying formalism as :

(..((n) :: {n:=0}, [* :({n:=n+1}, -]* .
)..)(n, m) :: {m:=0}, [-, {m:=m-1})]* {n=m} .

With this reformulation the treatment of '(..(' and ')..)' becomes part of the general treatment of the formalism.

Formal.

. - a β	--) --->	- a . β	
		and . - a β	, with a <> ")"
. - a β	-- a\`--->	. - a β	, with a\` <> ")"
. (β	-- (--->	(. β	
. :(β	-- :(--->	:(. β	
. a β	-- :(-) --->	. a β	(transition on the sequence of symbols ":(", 0 or more characters and a ")")

5.3 Boolean constructs

5.3.1 Boolean "and" between rules: cooperation

Informal.

We will discuss the method only for the case that no recursion in cf rules occurs. In that case it suffices to treat cooperation and Boolean "and" in compile-time. All possible paths are then represented by the items which are present in the itemset. Otherwise an equivalent treatment has to be added to the PTA, because different paths may be represented by items in different active L-nodes.

Suppose the grammar contains appearances of the cooperation symbol C (which may occur only in a rhs). The general form of a rhs with such an appearance is

α X{C} β .

Let A_C be the set of all items of the form α X{C}. β .

In section 2.2.6.1 we defined the following semantics of a cooperation: if in a created itemset I' an item of the form

α X{C} . β

is present then all items in A_C have to be present.

The algorithmic treatment is simple : after the construction of a new itemset all items with the dot behind the cooperation C have to be identified. These items have to be removed from the itemset when ($I' \cap A_C$) <> A_C .

Formal.

. X{C} β --X---> X{C}. β if ($I' \cap A_C$) = A_C

5.3.2 Boolean negation within a rule

Within a grammar rule a Boolean complementation can be expressed for a single symbol or for a regular expression. The symbol can be a terminal symbol, an intermediate symbol or a nonterminal. The discussion on the treatment of the complementation of a nonterminal and of a regular expression are related because, in general, a regular expression may be replaced by a nonterminal. Nonterminals and regular expressions generate, in general, strings of terminals. We are therefore confronted with the complementation of strings and with the problem how to treat it in the algorithm for the construction of LR-tables. We will develop a decomposition method for Boolean complementation together with the introduction of two new types of dots. The problem will be transformed into a problem where we treat individual characters and transitions of types of dots. In that context we will use the term "negation" instead of "Boolean complementation".

5.3.2.1 Negation of a single terminal

Informal.
In section 2.2.6.2 we defined the negation of a terminal as the complement of the set of which it is a member. The complement was taken with the aid of the set "UNIVERSE". For instance,

a\` = UNIVERSE - {a}.

The set a\` can further be treated as a set of range symbols. The negation X\` of a nonterminal X will be treated in a subsequent section, as far as the closure of X is concerned. After the reduction of such a closure item X or X\` will be returned. The father item has to react on these symbols. The rules are the same as for terminal symbols with no universe defined.

Formal.

```
. a` β  --a--->      ε
        --b--->      a . β    for all b ∈ UNIVERSE - {a}
. X` β  --X---> ε
        --X`--> X` . β
```

5.3.2.2 Negation of a string : 3 types of item dots

Informal.
In section 2.2.6.2 we defined the negation of a string x ∈ (T+I)* as the set of strings {w | w ∈ (T+I)*, |w| = |x|, w ≠ x}.
That is, x generates all strings w with the length of x, but unequal to x.
The function GOTO-ITEMSETS has as its second argument one symbol. If we want to extend the LR-table construction algorithm then we have to devise a method in which the string x\` can be broken up into individual characters.
In our explanation we will use the unifying formalism. For example the string x = "ab" (the concatenation of the characters "a" and "b") is written as :

x :: a, b.

The negation of x is then written as \`[a, b]1. If we use a separate "]" which belongs to a "\`[" then we will write "]\`". If we write items then we will, as usual, leave out the ", ".
We observe that

`[a, b]1 ≡ a`, * | a, b` .

Other possible decompositions are :

`[a, b]1 ≡ a`, * | *, b` .

and `[a, b]1 ≡ a`, b | *, b` .

The decompositions are easily verified by writing truth-tables. We choose the first manner of decomposition because of the disjunction of a and a`. For each symbol only one item will travel, .a` * or .a, b` (the dot will be refined later). In the other decompositions one or two items will travel. This will increase the number of itemsets that will be created.

The decomposition of the negation of a string of 3 characters is :

`[a, b, c]1 ≡ a`, *, * | a, `[b, c]1 .

In general, if

P_1 :: `[a, b, c, .., z]1 .

then we may write :

P_1 :: a`, $*^{(n-1)}$ | a, P_2 .
P_2 :: b`, $*^{(n-2)}$ | b, P_3 .
P_3 :: c`, $*^{(n-3)}$ | c, P_4 .

....

P_{n-1} :: y`, * | y, z` .

This schema may be read as :

situation P_1 : if an a` is read then treat the remaining characters as don't cares else (an "a" is read) go to situation P_2;

situation P_2 : if a b` is read then treat the remaining characters as don't cares else (a "b" is read) go to situation P_3; etcetera.

We will simulate this interpretation of the schema by the introduction of two new types of dots, the ".`" ("negdot") and the ".*" ("don't care dot"). The role of the .` is to follow all possible derivations of a Boolean complementation. Such a derivation is found when the .` strikes against the "]`" which corresponds with the "`[" where the ".`" originated. The role of a .* is to follow all strings with the same length as the derivations of a Boolean complementation, but which are unequal to such a derivation. This shadowing of the strings of equal length becomes effective when the last .` disappears. At that moment it is clear that no derivation will be found.

When a normal "." passes through a "`[" it changes to a ".`". The ".`" changes to a ".*" as soon as an input symbol does not match the item (this will be defined more precisely). When a ".`" strikes against the "]`" which corresponds with the "`[" where the ".`" originated then the item has to disappear. A ".*" will consume input symbols up to the "]`". Thereafter it will change to a normal "." .

Literally speaking, the dot types .` and .* are traveling towards the ]` which belongs to the `[where they originated.

Formal.

Is part of the formal treatment of the combination of Boolean negation and regular expressions and symbols.

5.4 Boolean negation and regular expressions

Informal.
For the treatment of the combination of Boolean negation and regular expressions it will be necessary to extend the concept of a dot. We define it as a triplet with a graphic representation.
A dot is a triplet (type, marker, brackets), where
- the *type* can be : normal dot ("."), neg dot (".`") and don't care dot(".*"),
- the *marker* is a label which is used in order to remember which items originated together from the passing of a normal dot through a negation bracket (for a normal dot there is no marker),
- *brackets* is the number of open negation brackets; for a normal dot this number is always 0; for a neg dot it is a single number, for a don't care dot it is a duplet (open active negation brackets, open passive negation brackets). The terminology will be clarified later on.
Example of a graphical representation : ".`$_m^b$" means a neg dot with marker m and b open negation brackets, ".*$_m^{b,\,c}$" means a don't care dot with marker m, b open active negation brackets and c open passive negation brackets.

Within a regular expression it is possible that after a "`[" a number of alternatives are present. According to De Morgan's law the negation of a number of alternatives has to be interpreted as the Boolean "and" of the negation of each of the alternatives. If any of the alternatives is matched then the whole expression between the negation-brackets is matched and recognition may not continue.
The foregoing is denoted in the following transformation :

.`[$\alpha$ | $\beta$ | ..]1 $\gamma$ => `[.`$_m^1\alpha$ | .`$_m^1\beta$ | ..]1 γ

The normal dot changes into a neg dot before each of the alternatives. The unique marker m is used in order to remember which items originated from the transformation. If one of the ".`$_m^1$" will strike against a "]`" then all items with dots which bear the same marker m have to disappear.
Negation brackets may be nested. A ".`$_m^b$" will change into a ".`$_m^{b+1}$" when it passes through a "`[".
A ".`$_m^b$" may, in the goto-function, change into a ".*$_m^b$". A ".*$_m^{1,\,0}$" will change into a ".$_m$" when it passes through a "]`". The marker m has to disappear as soon as the last ".`$_m^b$" changes into a ".*$_m^{b,\,0}$".
In order to deal rigorously with the combination of Boolean negations and regular expressions we have to treat all combinations of the 3 types of dots with the 4 brackets [, `[, ] and ]`. Moreover, the closing bracket "]" may appear in 4 appearances :],]1,]* and]+. However, it is possible to restrict the discussion by making use of the following observation. In section 2.2.6.2 we defined that the interpretation of `[$\alpha$}, where } means ], ]1, ]* or ]+, is the same as of `[[α}]1. For the treatment of a `[we therefore don't have to distinguish between the corresponding closing brackets.

We treated already the normal dot in combination with the [and the]. The other possible combinations are .`[, .*[, .`[, .``[, .*`[, .`], .*], .]`, .`]` and .*]` .

Opening brackets

Transformation of $`[\alpha\ \}$.
Formal:
$`[\alpha\ \} \quad => \quad `[\ [\alpha\ \}\ ]1 \qquad \text{for } \} =]\ ,\ ]1,\ ]*\ \text{ or }]+$

The case .[
Already treated.

The case .`[
The function of the neg dot is to follow all possible derivations. If it succeeds then all items with the marker m have to be removed. A normal open bracket indicates that a regular expression will start. The neg dot has to follow all possible paths within the regular expression. This is the same task as a normal dot has. We define therefore the same transformation rules as for the combination of a normal dot and a regular expression. The marker m will distribute itself over the items behind the [.
Formal:

$.`m^b[\alpha \mid \beta \mid ..]\ \gamma$	=>	$[.`m^b\ \alpha \mid .`m^b\beta \mid ..]\ \ .`m^b\ \gamma$
$.`m^b[\alpha \mid \beta \mid ..]1\ \gamma$	=>	$[.`m^b\ \alpha \mid .`m^b\beta \mid ..]1\ \ \gamma$
$.`m^b[\alpha \mid \beta \mid ..]*\ \gamma$	=>	$[.`m^b\ \alpha \mid .`m^b\beta \mid ..]*\ .`m^b\ \gamma$
$.`m^b[\alpha \mid \beta \mid ..]+\ \gamma$	=>	$[.`m^b\ \alpha \mid .`m^b\beta \mid ..]+\ \ \gamma$

The case .*[
The function of the don't care dot is to follow all possible input strings with the same length as a derivation, but not equal to a derivation. Therefore the don't care dot has to follow all possible paths within the regular expression. This is the same task as a normal dot has. We define therefore the same transformation rules as for the combination of a normal dot and a regular expression. The marker m will distribute itself over the items behind the [.
Formal:

$.*m^{b,\,c}\ [\alpha \mid \beta \mid ..]\ \gamma$	=>	$[.*m^{b,\,c}\ \alpha \mid .*m^{b,\,c}\ \beta \mid ..]\ \ .*m^{b,\,c}\ \gamma$
$.*m^{b,\,c}\ [\alpha \mid \beta \mid ..]1\ \gamma$	=>	$[.*m^{b,\,c}\ \alpha \mid .*m^{b,\,c}\ \beta \mid ..]1\ \ \gamma$
$.*m^{b,\,c}\ [\alpha \mid \beta \mid ..]*\ \gamma$	=>	$[.*m^{b,\,c}\ \alpha \mid .*m^{b,\,c}\ \beta \mid ..]*\ .*m^{b,\,c}\ \gamma$
$.*m^{b,\,c}\ [\alpha \mid \beta \mid ..]+\ \gamma$	=>	$[.*m^{b,\,c}\ \alpha \mid .*m^{b,\,c}\ \beta \mid ..]+\ \ \gamma$

The case .`[
This case concerns the creation of a negdot. Because of the transformation of $`[\alpha\}$ the negdot will further be distributed over the regular expression behind the `[.
Formal:
$.\ `[ \quad => \quad `[.`m^1$

The case .``[
The negation bracket passes through a `[. The bracket counter is incremented.
Formal:
$.`m^b\ `[\quad => \quad `[.`m^{b+1}$

<u>The case</u> .*`[

The don't care dot passes through a `[. Its task is simply to follow all strings with the same length as the strings which may be derived from the regular expression. If the don't care dot originated after the last `[the passive bracket counter c will be 0. When the don't care dot will encounter the]` on the current bracket level it will change into a negdot ("active") when b > 1. The counter c therefore acts as a guard to keep the don't care dot intact ("passive") as long as it travels through intermediate negation brackets.

Formal:

$.*m^{b,\,c}\ `[ \quad => \quad `[.*m^{b,\,c+1}$

Closure brackets

<u>The case</u> .`]

A neg dot encounters a normal closing bracket. The same transformation rules are followed as with a normal dot.

Formal :

$[\alpha \mid \beta.`m^{b} \mid ..]\ \gamma \quad => \quad [\alpha \mid \beta \mid ..]\ .`m^{b}\,\gamma$

$[\alpha \mid \beta.`m^{b} \mid ..]1\ \gamma \quad => \quad [\alpha \mid \beta \mid ..]1\ .`m^{b}\,\gamma$

$[\alpha \mid \beta.`m^{b} \mid ..]*\ \gamma \quad => \quad [.`m^{b}\alpha \mid .`m^{b}\beta \mid ..]*\ .`m^{b}\,\gamma$

$[\alpha \mid \beta.m^{b} \mid ..]+\ \gamma \quad => \quad [.`m^{b}\alpha \mid .`m^{b}\beta \mid ..]+\ .`m^{b}\,\gamma$

<u>The case</u> .*]

A neg dot encounters a normal closing bracket. The same transformation rules are followed as with a normal dot.

Formal :

$[\alpha \mid \beta.*m^{b,\,c} \mid ..]\ \gamma \quad => \quad [\alpha \mid \beta \mid ..]\ .*m^{b,\,c}\,\gamma$

$[\alpha \mid \beta.*m^{b,\,c} \mid ..]1\ \gamma \quad => \quad [\alpha \mid \beta \mid ..]1\ .*m^{b,\,c}\,\gamma$

$[\alpha \mid \beta.*m^{b,\,c} \mid ..]*\ \gamma \quad => \quad [.*m^{b,\,c}\,\alpha \mid .*m^{b,\,c}\,\beta \mid ..]*\ .*m^{b,\,c}\,\gamma$

$[\alpha \mid \beta.*m^{b,\,c} \mid ..]+\ \gamma \quad => \quad [.*m^{b,\,c}\,\alpha \mid .*m^{b,\,c}\,\beta \mid ..]+\ .*m^{b,\,c}\,\gamma$

<u>The case</u> .]`

cannot exist

<u>The case</u> .`]`.

A neg dot encounters a closing negation bracket. If it is the bracket on the level where it originated the b=1 and the negation has become effective. All items which are still traveling with the same marker have to disappear. If b > 1 then the negation was embedded within another negation. In that case the same strategy is followed as with the negation of a single character : the negdot changes into a don't care dot.

Formal :

$.`m^{b+1}\ ]` \quad => \quad]`\ .*m^{b,\,0}$

$.`m^{1}\ ]`\ \gamma \quad => \quad \varepsilon$, together with all items which bear the marker m

<u>The case</u> .*]` .

A don't care dot encounters a closing negation bracket. If c>0 then the dot traveled through passive negation brackets. If c=0 then the dot acts on the level where it originated and it has to change in a neg dot.

Formal :

.*m$^{b,\,c+1}$]`	=>	]` .*m$^{b,\,c}$
.*m$^{b+1,\,0}$]`	=>	]` .`m$^{b,\,0}$
.*m$^{1,\,0}$]`	=>	]` ·m

5.5 Boolean negation and symbols

The following rules are straightforward extensions of the discussions above.

5.5.1 Boolean negation and terminals

Formal :

.`m^{b} a β	-- a -->	a .`m^{b} β	
	-- b-->	a .*m$^{b,\,0}$ β	with b ∈ UNIVERSE-{a}
.*m$^{b,\,c}$ a β	-- . -->	a .*m$^{b,\,c}$ β	
.`m$^{b}$ a` β	-- a -->	a .*m$^{b,\,0}$ β	
	-- b-->	a .`m^{b} β	with b ∈ UNIVERSE-{a}
.*m$^{b,\,c}$ a` β	-- . -->	a` .*m$^{b,\,c}$ β	

5.5.1.1 Boolean negation and don't cares

Formal :

. * β	--a---> * . β	for all a ∈ (T+I)
.`m^{b} * β	--a---> * .`m^{b} β	for all a ∈ (T+I)
.*m$^{b,\,c}$ * β	--a---> * .*m$^{b,\,c}$ β	for all a ∈ (T+I)

5.5.1.2 Boolean negation and arb's

We defined the "=" in "=a" to generate the same strings as [a`]*. From this equivalence we may derive the following rules.

Formal :

.`m^{b} = a β	--a--->	.*m$^{b,\,0}$ = a .`m^{b} β
	--a`--->	.`m^{b} = a β
.*m$^{b,\,c}$ = a β	--b--->	.*m$^{b,\,c}$ = a .*m$^{b,\,c}$ β for all b ∈ (T+I)

5.5.1.3 Boolean negation and lines

We defined the "-" in "-a" to generate the same strings as [*]*. From this equivalence we may derive the following rules.

Formal :

.-	=>	.-.	for all dot types
.-	--.--->	.-	for all dot types

{These two rules imply the following rules :

. - a β	--a--->	. - a . β
	--a`-->	. - a β

.`m^{b} - a β	--a--->	.`$m^{b}$ - a .`m^{b} β	
	--a`-->	.`m^{b} - a .*$m^{b,\,c}$ β	
.*$m^{b,\,c}$ - a β	--b--->	.*$m^{b,\,c}$ - a .*$m^{b,\,c}$ β	for all b ∈ (T+I)

}

5.5.1.4 Boolean negation and ranges of terminal symbols

Informal :
In section 5.2.4 we discussed that ranges of terminal symbols are treated by the creation of a new set of range terminals which are not overlapping. On these new terminals all the formal rules are applicable.

5.5.1.5 Boolean negation and tree symbols

Formal :

.`m^{b} - a β	--)--->	- a .`m^{b} β	this rule restricts the rule for lines
	and	.`m^{b} - a β	, when a <> ")"
.`m^{b} - a β	--a`--->	.`m^{b} - a β	, when a` <> ")"
.`m^{b} a β	-- :(-) --->	.`m^{b} a β	(transition on the sequence of symbols ":(", 0 or more characters and a ")")
.*$m^{b,\,c}$ - a β	--)--->	- a .*$m^{b,\,c}$ β	
	and	.*$m^{b,\,c}$ - a β	, when a <> ")"
.*$m^{b,\,c}$ - a β	--a`--->	.*$m^{b,\,c}$ - a β	, when a` <> ")"
.*$m^{b,\,c}$ a β	-- :(-) --->	.*$m^{b,\,c}$ a β	(transition on the sequence of symbols ":(", 0 or more characters and a ")")

5.5.2 Boolean negation and nonterminals

Informal.
In section 2.2.6.2 we defined that the negation of a nonterminal generates all strings w with length up to the length of the largest string that can be generated by B, but with $x \neq w$, where B =*> x. The strings that can be generated by B are the strings which can be generated by the closure items of B.
In the function CLOSURE for each item A :: α.Bβ and each production B :: γ the item B :: .γ is added (if not already present).
This rule has to be extended for the item A :: α.B`β and for the dot-types ".`" and ".*". We will treat therefore the combinations .B`, .`B, .*B, .`B` and .*B` . The treatment concerns the adding of a closure item and the reduction of an item.

We already formulated that the shifting of nonterminals obeys the same rules as the shifting of terminals:

α.B`β	--B`-->	α.B`β
	--B -->	ε

The symbols B and B\` have to be delivered by the reduction of a closure item of B. If such a closure item is B :: γ then we will add as a closure of B\` the item B :: .\` γ. If a derivation of B will be found then a reduce item B :: γ .\` will be constructed and we will return a B. If a derivation is not found then a reduce item B :: γ .* will be constructed and we will return a B\`.
The function of the negdot includes the function of a normal dot : it follows all possible derivations. But it will also signal a failure when it changes into a don't care dot. The function of a don't care dot is the most limited one : it only follows strings with the correct length. We will follow the strategy to use in a closure item the type of dot that has a function that is just "strong enough" to supply, after a reduction, to the father item all possible continuations.
From that strategy the following table originates.

on father item	add closure item(s)	after reduction of the item	reduce symbol	shift on father item will be
A -> α . B β	B -> . γ	B -> γ.	B	A -> α B .β
A -> α . B\` β	B -> .\`$_n{}^1$ γ	B -> γ.\`$_n{}^1$	B	ε
		B -> γ.*$_n{}^{1,\,0}$	B\`	A -> α B\` . β
A -> α .\`$_m{}^b$ B β	B -> .\`$_n{}^1$ γ	B -> γ.\`$_n{}^1$	B	A -> α B .\`$_m{}^b$ β
		B -> γ.*$_n{}^{1,\,0}$	B\`	A -> α B .*$_m{}^{b,\,0}$ β
A -> α .\`$_m{}^b$ B\` β	B -> .\`$_n{}^1$ γ	B -> γ.\`$_n{}^1$	B	A -> α B\` .*$_m{}^{b,\,0}$ β
		B -> γ.*$_n{}^{1,\,0}$	B\`	A -> α B\` .\`$_m{}^b$ β
A -> α .*$_m{}^{b,\,c}$ B β	B -> .*$_n{}^{1,\,0}$ γ	B -> γ.*$_n{}^{1,\,0}$	B\`	A -> α B .*$_m{}^{b,\,c}$ β
A -> α .*$_m{}^{b,\,c}$ B\` β	B -> .*$_n{}^{1,\,0}$ γ	B -> γ.*$_n{}^{1,\,0}$	B\`	A -> α B\` .*$_m{}^{b,\,c}$ β

The writing of A :: α.B\`β is equivalent to the writing of A :: α.\`[B]1 β . If we imagine that we replace B by all its possible rhs's (written as alternatives of each other, i.e. χ | δ | ..) then we can relate the treatment of .B\` to the treatment of the negation of regular expressions, and vice versa.
The relation concerns the transformation of a dot when it passes through an opening and a closing bracket in a regular expression and the addition of a closure item respectively the reduction of an item for a nonterminal. The reader may verify that the rules are in correspondence with each other.

5.6 Boolean negation and Boolean "and" between rules

After a new itemset is created two checks have to be performed which stem from the treatment of the Boolean operators within and between rules:
- the removal of items with markers which are not longer allowed (a "type-M item")
- the removal of items which reached their cooperation symbol and which did not meet all the other items with the same cooperation symbol (in accordance with section 5.3.1). These items are not longer valid. (A "type-C item").

It is possible that a type-M item is also a type-C item. When this item is removed the cooperation is not longer valid, and its associated type-C items have to be removed also. This process is iterative but will, of cause, terminate. The new itemset will eventually become empty.

This finishes the extensions to the algorithms for the construction of an LR-table for all the sub-formalisms within the unifying formalism. In the next chapter we will concern ourselves with an efficient implementation of the extended algorithms within a compiler.

6. A compiler

6.1 Overview

The task of the compiler is to transform a grammar, which is written in the unifying formalism, into a program for the PTA. In this chapter we will discuss the different stages of the transformation. Before we do so we will extend the algorithm still further with a new method for the creation of a FSA for left- and/or right recursive cfg's.
In chapter 5 we discussed the extensions for a number of the sub-formalisms. The extensions for the remaining sub-formalisms are treated in this chapter. They concern : the treatment of variables, reports and builds, the adjustment for the closure algorithm in the case of transduction rules and cascaded grammars.
During the development of a grammar the grammar writer will often make use of the compiler. The complexity of the runtime of the compiler is therefore important. In order to improve it we calculate a number of sets and relations. With the aid of these sets and relations the extended algorithm for the creation of LR-tables may be rephrased into a more efficient one.
Some of the constructed datastructures are transported to the runtime system in order to enable a conversation with the user in terms of the original grammar.

The compiler consists of the following modules :

- "Readgrammar"	:	reading in the grammar and storing it in a datastructure
- "Preparesets"	:	calculating from this datastructure a number of sets and relations
- "GenLRsets"	:	construct itemsets
- "Supscanner"	:	construct a scanner-table for an itemset
- "SupLRcode"	:	construct code for a generated itemset of a lhs or a rhs.

The compiler is complemented by a linking loader for cascaded grammars and a disassembler for the generated code.

In chapter 2 we mentioned some important topics which play a role in the literature on LR-parsing. They will be dealt with in the following modules :
Readgrammar :
- no transformations of the input-grammar
GenLRsets :
- the elimination of unit reductions
- the treatment of empty rules
- the handling of shift/reduce-reduce conflicts
- the minimization of the number of generated states
SupLRcode :
- generation of stack instructions (only if necessary)
- the generation of shared code.

The user may influence the behaviour of the compiler by the setting of switches. We list them together with their function :

- switches for the restriction of the type of automaton for which code has to be generated :

- - "multi", if false : if no cf-rules exist with recursion then do not generate the instruction LSTK which is responsible for the creation of nodes in the L-dag but generate instead the instruction NLSTK; in that way code is generated for a FSA
- - "test_comp_sets", if true : when multi = false then if cf-rules exist with only left- or right-recursion then optimize in such a way that still a FSA will be produced (to be described later)

- switches for the disambiguation for ambiguous type-0 or transduction grammars; a number of options may be selected to resolve conflicts in inadequate states for rules with |lhs|>1:
- - "shift_no_reduce" : in case of a shift/reduce conflict in the LR-table: precedence of shift
- - "reduce_no_shift" : in case of a shift/reduce conflict in the LR-table: precedence of reduce(s)
- - "one_cs_reduce" : in case of a reduce/reduce conflict between two or more rules : precedence of a rule with a longer rhs; if the length is the same : the rule which was placed first in the grammar

- switch for transduction purposes :
- - "add_cs_rules" : add to each generated itemset all type-1 and -0 items with the dot in the first position of the rhs (if not already present)

- switch for optimization purposes :
- - "intermediate_unit_reduction" : (don't) generate code (the instruction PLRDC) to store in a parse a nonterminal which originates from a unit reduction

- switch for Boolean constructs :
- - "shared_universe" : add the set of intermediate symbols I to the universe

- switch for impatient writers of type-2 grammars :
- - "interactive" : use the compiler as an Earley-parser

- switch for monitoring purposes :
- - a number of switches which effect the printing of the datastructures which are created in the different modules.

It is not possible to discuss each module in full detail. In their implemented form they consist together of about 13.000 lines of Pascal code (including comments). The optimization of the treatment of empty rhs's and unit rules complicates the treatment of the extensions of the LR-algorithm. This will be dealt with in forthcoming publications of M. Elstrodt and the author. However, it is possible to discuss the main algorithms in a more simplified form and to indicate how they are extended towards their completed form.

6.2 Reading the grammar : module "Readgrammar"

In the module "Readgrammar" a grammar which is written in the unifying formalism is read in and is stored in a datastructure from which the module "Preparesets" will calculate all necessary sets and relations.
We adhere to the principle that in the communication between the grammar writer and the compiler and the runtime system the original rules have to be used. This is the reason why we, for instance, do not transform regular expressions into sets of recursive cf rules which describe the same (sub)language ("weakly equivalent grammars"). In the communication

with the user it is in principle possible to translate such a transformed grammar rule back into its original form, but in practice this is a complicated affair.
In chapter 2 we gave the definition of a U-grammar. It concerned a number of sets of symbols and a set of rewriting rules. The metagrammar describes the syntax of the rewriting rules. With the aid of the metagrammar a U-grammar can be parsed and transformed into a datastructure. This is the task of the module "Readgrammar". Together with the construction of the datastructure some useful indexes in the datastructure are built.
Thompson (1968) described an algorithm for the transformation of a grammar which contains regular expressions into a datastructure. We will first present the principle of a modification of Thompson's algorithm. Then we will discuss the actual analysis of the grammar. Lastly we will present the datastructures which are in use in the module "Readgrammar".

6.2.1 From grammar rules with regular expressions to a datastructure

As we mentioned in chapter 2, regular expressions in cfg's are treated by Madsen and Kristensen (1976), Purdom and Brown (1981), Heilbrunner (1979), Lalonde (1979 and 1981). All these authors are concerned with the maintenance of the LR(k) property. Some of them transform an ecfg into a cfg (like we did in an earlier version of our parser), others generate stack instructions in order to be able to inspect the stack. With normal cfg's the size of a rhs determines how many symbols have to be popped off the stack in order to make the returnstate visible. With ecfg's this is not possible : one does not know beforehand how many symbols will be covered by a rhs. The approach of some authors is that the number of symbols can be recovered during runtime with the aid of extra bookkeeping instructions.
In our approach we will rely on the PTA which keeps with each item its returnstate(s). When an item reduces it will simply return to that (those) returnstate(s).
Therefore we concentrated in chapter 5 on the extension of the LR-table construction technique for regular expressions in a lhs or a rhs of a grammar rule. We introduced there the idea of the transformation of instable items. The algorithm for the construction of a datastructure out of a regular expression will take care of this transformation. It can also be seen as a translation of a grammar written in the unifying formalism into a syntax-diagram or into an ATN-network (when also actions are used which contain operations on variables).
There have been several approaches to the problem of the practical construction of a FSA for a regular expression. In (Aho, Sethi, Ullman, 1986) a number of them are recapitulated. One approach is to construct a NFA, transform it into a DFA and then minimize the DFA. Another approach constructs a DFA directly out of the syntax tree of the syntactically analyzed regular expression. This is done by calculating a number of functions from the syntax-tree. One of these functions gives e.g. the positions in the regular expression which may follow after a transition on the current position. With the aid of a state-construction method which makes use of the calculated functions the NFA is build. This method is akin to the LR construction method if one relates positions in a regular expression to items in grammar rules.
A grammar in the unifying formalism consists of a number of rules which contain regular expressions. We use the method of state construction for the construction of the parser as a whole. For each side of a grammar rule we construct a NFA. From these NFA's we calculate a number of useful functions and relations.
In (Aho, Sethi, Ullman, 1986, p. 122) Thompson's method for the construction of a NFA out of a regular expression is discussed. The principal idea is to use a building block for each pair of brackets and to build larger blocks from smaller ones. Entries and exits of blocks may be connected to each other. We use the same approach (which is also used in the Unix utility Lex), but with the following differences :

(a) we have to take into account the expressions for the other sub-formalisms,

(b) we do not construct the blocks off-line from the information in a parse tree but we build the blocks on-line, in parallel with the on-line recognition of the grammar itself (which is performed by the PTA); the on-line construction is motivated by the possible speedup of the compiler when executing on parallel hardware.

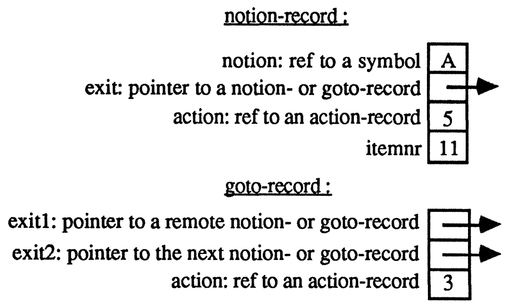
In the unifying formalism a number of sub-formalisms may be written within the action-brackets. We called them "actions". The actions may be placed virtually everywhere within a regular expression and our algorithm has to take this placement into account. Most of the actions themselves will be translated directly into code for the PTA and will be stored as a subroutine. Only the sub-formalism of cooperations will influence the behaviour of the algorithm Construct-itemsets. In the datastructure for the storage of grammar rules a reference to a subroutine has to be stored in the proper place. In our examples we will indicate that place. To concentrate the discussion on this aspect we will assume that only one kind of action may be present : the "report". This action will be representative for all other possible actions.

6.2.1.1 The principle of the datastructure for the storage of grammar rules

The datastructure for the storage of a rule with regular expressions and actions is built with four kinds of records : a "dope-record", a "notion-record", a "goto-record" and an "action-record". In the first one the two references to the datastructures for a lhs and a rhs are kept. In the second one a reference to a notion may be stored, in the third one two pointers are present which pave the road to successor notion-records, and in the fourth one all necessary information concerning actions is stored. We will leave out the fourth one in our discussion. For the purpose of illustration in examples we write, if appropriate, a reportnumber instead of a pointer to a full action-record by way of abbreviation. Items are identified as positions immediately before a notion-record and are stored as an integer in that record. Successor items will be found when all possible roads are followed through connected goto-records. A road ends on a notion-record. All actions which are found on the road are associated with this shift from an item to a successor item (including the starting notion-record, excluding the successor notion-record).

The two most important records in a picture :

notion-record = (notion, exit, action, itemnr)
goto-record = (exit1, exit2, action)


notion-record :
notion: ref to a symbol
A
exit: pointer to a notion- or goto-record
action: ref to an action-record
5
itemnr
11
goto-record :
exit1: pointer to a remote notion- or goto-record
exit2: pointer to the next notion- or goto-record
action: ref to an action-record
3

We concentrated more on the on-line construction of the resulting datastructure than on a full optimization of it. It is possible to replace some combinations of resulting notion- and goto-records by a combination with fewer records. This is exemplified in the figure of the first example. Two goto-records G1 = (0, ^G2, 0) and G2 = (P1, P2, 0) can there be merged into one goto-record (P1, P2, 0). In our case little will be gained by such an optimization.

The end of a side of a grammar rule is represented by a notion-record which contains no pointer to a notionname. In the case of a rhs the exit of this notion-record points to the entry of the corresponding lhs, in order to have a reference to the start of the corresponding first item of the lhs when a rhs reduces. In the case of a lhs this pointer is nil.

The essential idea is the identification of blocks in a grammar rule which correspond with a regular expression.
A block may contain other blocks. A block has one entrance and may have multiple exits. During the construction of a block its entrance is known, but not its exits. Therefore a list is maintained with pointers to all places where the pointer to the next block has to be filled in (the standard technique for handling forward references). Because regular expressions may be nested we keep a stack for all the information which concerns the construction of the blocks.

We have to construct the essential blocks for
1. notions
2. alternatives
3. blocks between brackets.

Ad. 1 Notions.
A notion will be represented by a single notion-record.

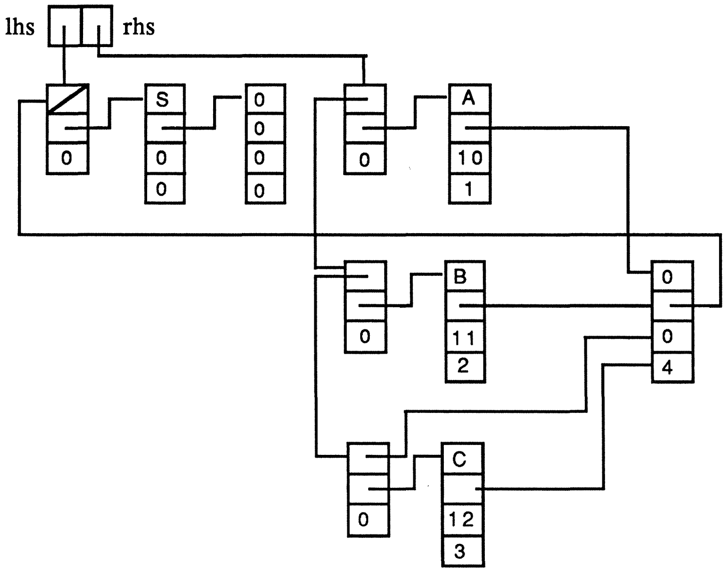
Ad. 2 Alternatives.
Each alternative is preceded by a goto-record. One pointer of the goto-record points to the first record of this alternative, another one will point to the goto-record of the following accompanying alternative.
In order to be consistent a whole side which has no alternative will be preceded also by a goto-record.
The exits of blocks which represent mutual alternatives will point to the same record, whether it is a notion- or a goto-record.

Ad. 3 Blocks between brackets.
A block between square brackets is treated as a set of alternatives. If the alternatives are placed within brackets followed by a "*" (zero or more occurrences) or a ε (zero or one occurrence) then the alternative-pointer of the last goto-record before the block of the last alternative will point to the next block.

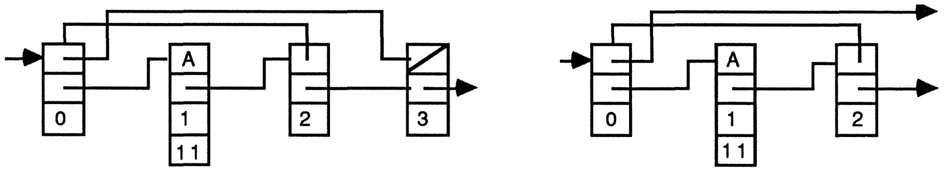
Examples.

The examples concern combinations of actions with
- ad. 2. alternatives :
- - ex. 1: a complete grammar rule S :: A{R:10} | B{R:11} | [C{R:12}].
- ad. 3. the 4 basic constructions of regular expressions, written as an infix of a side of a rule :
- - ex. 2: , [A{R:1}]{R:2}*{R:3}, and , [A{R:1}]{R:2}*,
- - ex. 3: , [A{R:1}]{R:2}+{R:3}, and , [A{R:1}]{R:2}+,
- - ex. 4: , [A{R:1}]1{R:2}, and , [A{R:1}]1,
- - ex. 5: , [A{R:1}]{R:2}, and , [A{R:1}],

Example 1: S::A{R:10}|B{R:11}|[C{R:12}].
If S is the startsymbol then the itemnumbers 1, 2, 3 and 4 are associated.

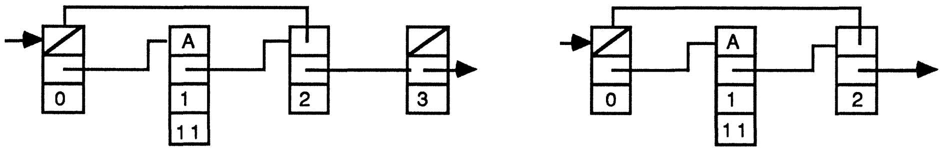


Example 2: , [A{R:1}]{R:2}*{R:3}, and , [A{R:1}]{R:2}*, (suppose itemnr. 11 is associated with notion A)



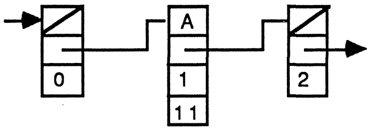
If {R:2} is absent the structure is the same, but the second goto-block will not contain a pointer to an action-record.

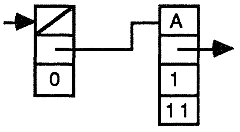
Example 3: , [A{R:1}]{R:2}+{R:3}, and , [A{R:1}]{R:2}+, (suppose itemnr. 11 is associated with notion A)



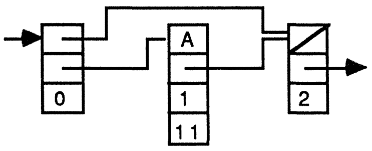
If {R:2} is absent the structure is the same, but the second goto-block will not contain a pointer to an action-record.

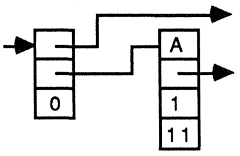
Example 4: , [A{R:1}]1{R:2}, and , [A{R:1}]1, (suppose itemnr. 11 is associated with notion A)





Example 5: , [A{R:1}][R:2}, and , [A{R:1}], (suppose itemnr. 11 is associated with notion A)





6.2.1.2 The on-line algorithm

The algorithm for the construction of the datastructure for a grammar rule which contains regular expressions reacts on the occurrence of the following symbols:

- start of a grammar rule
- end_of_rule_sign (the end of a grammar rule, by default denoted by a '.')
- rewrite_sign (the separation of a lhs and a rhs, default '::')
- notion
- action (between action_brackets, default '{' and '}')
- continuation_sign (between two consecutive notions, default ', ')
- alternative_sign (indication of an alternative, default '|')
- open_regexpr (the opening square bracket of a regular expression, default '[')
- close_regexpr (the closing square bracket of a regular expression, default ']')
- zeromore, onemore and one (behind a close_regexpr, default '*', '+', '1'), sometimes called "indicators".

The lhs and the rhs of a grammar rule are treated in the same way.

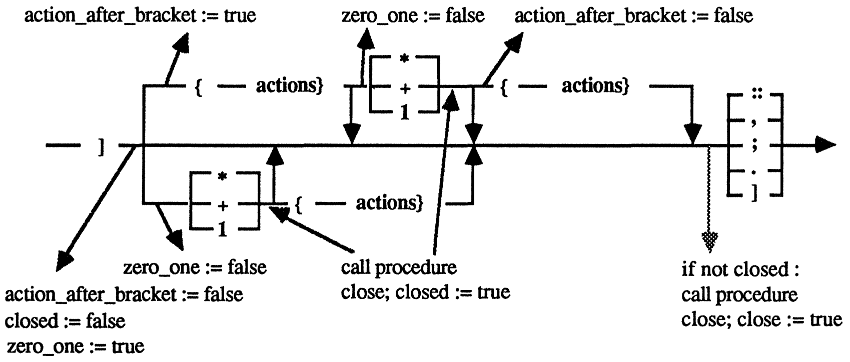
The algorithm is constructed along the following main lines.
During the on-line construction the integrity of the current block has to be kept. The nesting of blocks will be reflected by the use of a stack. The current block is referenced by the topmost stack-record. A stack-record consists of the 4 fields :
- T0: the indication of the reason why this record is on the stack :
  - '#': start of a lhs or a rhs
  - '|': new alternative
  - '[': nesting of a new block,

(A property of the algorithm is that, within one block, only one T0 with a '|' will be on the stack.)
- T1: a pointer to the first and only entry to the block, which is a goto-record (it has to be kept because in the case that the block is ended with a zeromore or a onemore a pointer has to be placed from the last goto-record to the first one),
- T2: a local help which remembers the address of the exit2-field in a goto-record at the start of a block where a pointer to the next alternative of the current block has to be filled in,

- T3: a list of elements in which the exits of the current block are sampled (these are addresses of exit-fields within a notion- or a goto-record).

We assume that the input is written correctly (as described by a U-grammar). After the reading of each character an action is performed with no backtracking.
After the treatment of a character the integrity of the current block has to be intact. The treatment of each character consists therefore of two parts:
- extend the current datastructure in a minimal way in order to reflect the structure of the grammar rule up till now
- keep the integrity of the block in order to be prepared for further processing.

A block starts with the start of a lhs or rhs, with an alternative_sign or with an open_regexpr. This usually implies a push of the stack, unless the current stack-record can be used again.
The current block will be ended by the characters : alternative_sign, close_regexpr, rewrite_sign and end_of_rule_sign. This ending is handled by the procedure CLOSE and usually implies the popping of the stack, with a transport of T3. When actions occur directly after a close_regexpr this is registered by the switch actions_after_bracket with the value true. In that case the popping is postponed because it is not known whether a zeromore or onemore will follow. This postponement of the closing of a block is indicated by the switch "close" with the value false. Upon the reading of the other closing characters and the continuation sign the postponed pop is executed.
The reading of the remaining characters implies a continuation within a block. Special care is needed for the characters zeromore and onemore in combination with actions : the datastructure has to be built according to the semantics which were discussed in chapter 2. The absence of zeromore, onemore and one implies "zero or one" which is indicated by the switch "zero_one" with the value true.
The setting of the switches is recapitulated in the following syntax-diagram which depicts the possible situation after a close_regexpr. To be short the default symbols are used.



The data elements are specified within the unifying formalism. The algorithm is presented in a pseudo Pascal, where we assume that we can manipulate with the addresses of individual fields of records. If P is such an address then we denote by P[] its content.

notion_record :: (notion, exit, action, itemnr). { see before }
goto_record :: (exit1, exit2, action). { see before }
stack_record :: (T0, T1, T2, T3). { see before }

```
T0 :: '#' | '|' | '['  .
T1 | T2 :: ^goto_record.
T3 :: [^goto_record.exit1 | ^goto_record.exit2 | ^notion_record.exit]+ .
dope_record :: (lhs, rhs).
lhs | rhs :: ^goto_record. notion :: ^symbol. exit :: ^ng_record. action :: ^action_record.
ng_record :: notion_record | goto_record.
exit1 | exit2 :: ^ng_record.
action_record :: (-). { not further specified; contains everything that can be denoted as a
                        reporting action }
NR :: notion_record. AR :: action_record. GR :: goto_record.
dope :: dope_record. stack :: [ stack_record ]*.
symbol_before | symbol :: ℂ . stackpointer | nitem | itemnr :: ℕ.
zero_one :: 𝔹 .
leftside:: 𝔹. { if true : we are reading in a lhs }
closed :: 𝔹. { if true : a block is constructed and is referenced by the topmost record of the
                stack }
```

```
Program REX
- stackpointer := 0; nitem := 0; closed := zero_one := true; action_after_bracket := false
- put an empty record on the stack
- START_OF_A_GRAMMAR_RULE
- while not end-of-file do
   - read symbol
   - case symbol of
     - NOTION:
        - allocate notion-record (NR)
        - if leftside: NR^.itemnr := (nitem := nitem + 1)
        - lookup notion in symboltree, place reference in NR^.notion
        - for all P in T3 do
           - P[] := NR
        - T3 := address(NR^.exit)
     - CONTINUATION_SIGN :
        - if not closed: CLOSE(postponed)
     - ALTERNATIVE_SIGN :
        - if not closed: CLOSE(postponed)
        - if T0 = '|' then POP           { merge 2 alternative blocks into one }
        - allocate goto-record (GR)   { as start of a new alternative block }
        - T2[] := GR              { link from starting goto-record of current block }
        - T2 := address(GR^.exit1)  { overwrite starting goto with new one }
        - PUSH('|', 0,  0,  address(GR^.exit2) )     { only '#' and '[' indicate a (nested)
                sequence of alternatives; with '|' the start and finish of such a
                sequence do not have to be remembered in T1 and T2 }
     - OPEN_REGEXPR :
        - allocate goto-record (GR)   { as start of a new block }
        - for all P in T3 do
           - P[] := GR
        - T3 := 0
        - PUSH( '[', GR, address(GR^.exit1, address(GR^.exit2) )
     - CLOSE_REGEXPR :
        - if not closed: CLOSE(postponed)   { close the last alternative, if not done }
        - if T0 = '|' then POP          { if alternative, merge with former one }
        - action_after_bracket := closed := false; zero_one := true    { initialise }
      - ZEROMORE :
        - zero_one := false
        - CLOSE(zeromore)   { close current alternative and make pointers for zeromore }
      - ONEMORE :
        - zero_one := false
        - CLOSE(onemore)    { close current alternative and make pointers for onemore }
      - ONE :
        - zero_one := false
        - CLOSE(one)                    { close current alternative }
      - OPEN_ACTION_BRACKET :
        - allocate action-record (AR)
        - if closed
          - if symbol_before = notion
            - NR^.action := AR
          - else
            - allocate gotorecord(GR)
            - GR^.action := AR
            - for all P in T3 do
              - P[] := GR
            - T3 := address(GR^.exit2)
        - else
          - if symbol_before = close_regexpr :
            - action_after_bracket := true
          - fill in actions in AR and read until close_action_bracket
      - REWRITE_SIGN :
        { closing a lhs }
        - if not closed: CLOSE(postponed)
        - if T0 = '|' then POP
        - allocate notion-record(NR)  { for the marking of the end of the lhs }
        - for all P in T3 do
          - P[] := NR
        { starting a rhs }
        - leftside := false
        - allocate gotorecord (GR)
        - DOPE.rhs := GR
        - T0 := '#'; T1 := GR; T2 := address(GR^.exit1);  T3 := address(GR^.exit2)
        - closed := true
      - END_OF_RULE_SIGN :
        - if not closed: CLOSE(postponed)
        - if T0 = '|' then POP
        - allocate notion-record (NR)          { for the marking of the end of the rhs }
        - NR^.exit := pointer to start of lhs
        - NR^.itemnr := (nitem := nitem + 1)
        - for all P in T3 do
          - P^.exit2 := NR
        - if not end_of_file
          - START_OF_A_GRAMMAR_RULE
{ end of rex }
```

```
procedure PUSH(T0, T1, T2, T3)
   - stackpointer := stackpointer + 1
   - stack[stackpointer] := (T0, T1, T2, T3)
( end of push }

procedure POP
   - stack[stackpointer-1].T3 :=  stack[stackpointer-1].T3 || stack[stackpointer].T3
         { '||' : append the two lists }
   - stackpointer := stackpointer - 1
( end of pop }

procedure CLOSE(indicator)           { this procedure completes a block }
  { a goto-record is needed when
         - actions have to be stored
         - and/or a pointer for repetition has to be stored
         because of the 'and' we serve with this procedure two purposes }
- if action_after_bracket or (indicator ∈ {zeromore, onemore} )
   - allocate goto-record (GR)
   - if action_after_bracket
      - GR^.action := AR
      - action_after_bracket := false
   - for all P in T3 do
      - P[] := GB
   - T3 := address(GR^.exit2)
   - if indicator ∈ {zeromore, onemore}
      - GR^.exit1 := T1
- if (indicator = zeromore) or zero_one
   - T3 := T3 !! T2      { save the continuation-pointer T2 of the last alternative
                                   as an exit }
- POP
- closed := true
{ end of close }

procedure START_OF_A_GRAMMAR_RULE
- leftside := true
- allocate DOPE : a dope-record for a field, which has 2 fields, pointing to the lhs and rhs
- allocate goto-record (GR)
- DOPE^.lhs := GR
- T0 := '#'; T1 := GR; T2 := address(GR^.exit1); T3 := address(GR^.exit2)
- closed := true
{ end of start_of_a_grammar_rule }
```

6.2.2 Parsing of grammar rules according to the metagrammar

We use the compiled metagrammar in order to parse a user grammar. (The inherent bootstrapping problem did not manifest itself because of the evolution process of the Parspat compiler and runsystem). In the former section the algorithm reacted on the occurrence of a number of symbols. This can be achieved by denoting reports in the metagrammar at the appropriate places. As we have seen with the PTA, the reports are brought to the outside world as soon as possible (after the resolution of temporal ambiguities).

6.2.3 Implementation : Datastructures to be built

6.2.3.1 Symbol Trees

The names of symbols are stored in symbol trees. The symbols get an identification number. For each tree an index is built: given an identification number a pointer is returned to the corresponding name. This is done by making use of standard techniques for the maintenance of symbol tables.

According to the definitions of chapter 2 the notions in a U-grammar can be divided in the following categories. We list them in the order in which they get their identification number:
- the set of reserved symbols
- the set of nonterminals N (which were defined as the symbols that are rewritten, as only symbol, at a lhs), subdivided in
- - the startsymbol (in the case of a PSG)
- - the remaining nonterminals
- the set of intermediate symbols I
- the set of terminal symbols T, subdivided in
- - lexicon notions : these are prefixed by a "$"
- - notions which generate an element of a set denoted by a range of characters
- - notions with the length of 1 character which generate that character.

6.2.3.2 The other datastructures

In the implementation of the compiler the goto-, notion- and action-records, of course, contain more fields than shown in the algorithm because of the presence of the other sub-formalisms, like : negation, cover symbols and formal parameters (variables and cooperations).

6.3 Preparing useful sets and relations : module "Preparesets"

In chapter 5 we extended the functions "Goto-itemset" and "Closure". This resulted in
- the definition of traveling rules for items with 3 types of dots
- the transformation of items
- the removal of items from itemsets in the case of the use of cooperations and/or negations
- the addition of rules for the adding of closure items in the case of negations.

In the module Preparesets we will prepare a number of useful sets in order to construct the itemsets in a more efficient way than is indicated in the 4 functions for LR table construction.

The shift rules for items can be expressed as triples (item, symbol, item), the reduce rules as pairs (item, symbol). We defined an item itself in section 5.4 as a triplet (type, marker, brackets), where
- the *type* can be : normal dot ("."), neg dot (".'") and don't care dot (".*"),
- the *marker* is a label which is used in order to remember which items originated together from the passing of a normal dot through a negation bracket (for a normal dot there is no marker),
- *brackets* is the number of open negation brackets; for a normal dot this number is always 0; for a neg dot it is a single number, for a don't care dot it is a pair (open active negation brackets, open passive negation brackets).

Each item has a unique label. In our implementation it is an integer which counts for each notion in a grammar rule three times (normal dot, neg dot and don't care dot), ordered from the first rule to the last rule. This ordering is arbitrary. The values within the triplet of an item are related to the position of the itemdot in the grammarrule, and therefore to the itemnr. For a unique characterization of the item they can be left out. The marker, however, can take a number of values which will depend upon the path that is followed up to the current itemset. The conclusion is that an item can be uniquely characterized by a pair of integers (itemnumber, marker). Therefore, within one itemset there may be present a number of items with the same itemnumber, but with different markers. The traveling rules for the item itself will not depend upon the marker. The marker plays only a role when related items have to be removed. It is therefore possible to express the traveling rules as triples (itemnumber, symbol, itemnumber) and to keep for each itemset a bookkeeping for the markers which are associated with the items. We will therefore keep the handling of the markers out of our discussion.

In the algorithm GenLRsets of section 6.4.5 for the construction of itemsets the following datastructures are used:

datatype :	name :	function :
array of set of items	items_before_notion[notion]	gives for each notion the items where the itemdot precedes that notion
array of set of items	follow_items_of_item[itemnr]	gives for each item its item_projectives
array of integers	notion_behind_item[itemnr]	gives for each item the notion behind its itemdot
set of items	start_items_before_ nonterminals	gives all starting items with the dot before a nonterminal
set of items	start_cs_items	gives all starting items of type-1 and -0 rules
set of items	nonterminals_after_start_ or_middle_items	gives all non-reducing items with the dot before a nonterminal
set of items	all_cf_reduce_items	gives all cf reducing items at a rhs
set of items	all_cs_reduce_items	gives all type-1 and -0 reducing items at a rhs
array of set of items	closure_items_of_ notion[notion]	gives for a notion the numbers of the start-items at the rhs with that notion as the first symbol of the lhs

These sets are derived in a straightforward manner by a depth-first walk through the datastructure for grammar rules which was created in the module Readgrammar. Closure_items_of_notion is constructed with the aid of an adapted version of the closure-algorithm of Warshall (1962). The adaptation concerns negated nonterminals and type-1 and -0 rules.

6.4 Construct Itemsets : module "GenLRsets"

After the construction of the sets in the module "Preparesets" it is now possible to specify the algorithm for the construction of itemsets. It works in principle for a rhs as well as for a lhs.

6.4.1 Generating a FSA for left- and right-recursive cfg's

In every shift/reduce parsing automaton the reduction of a rhs of a rule takes more elementary operations than a shift operation. It seems therefore appropriate to avoid as much as

possible the generation of reduce instructions. A reduce instruction may be expected to be generated for each alternative of the rewriting of a nonterminal.
In general, nonterminals are used by a grammar writer for
(a) the abbreviation of common substrings in rhs's
(b) the expression of recursion.
We will discuss how the algorithm for the construction of itemsets can be extended in order to generate only shift entries in the LR-table for cfg's which do not contain infix recursion.

(a). Conceptually, a nonterminal which is used for the abbreviation of common substrings in a rhs may be replaced by its rhs's. We will simulate this replacement by a combination of Earley's parsing algorithm with the LR parser-generation algorithm. Earley's algorithm was recapitulated in section 3.6 . We defined an Earley-item as a tuple $<A::\alpha.\beta, f>$ where $A::\alpha\beta$ is a production and f is an integer, representing the number of the itemset where the closure item $A::.\alpha\beta$ originated (itemsets are numbered in the sequence in which they are created). In the sequel we will call itemset f the "father itemset" of item $A::\alpha.\beta$. Moreover, we will denote the set of Earley-items $\{<A::\alpha.\beta, f_1>, <A::\alpha.\beta, f_2>, ..., <A::\alpha.\beta, f_n>\}$ by $<A::\alpha.\beta, \{f_1, f_2, ..., f_n\}>$. The set $F=\{f_1, f_2, ..., f_n\}$ will be called the "set of fathers" of item $A::\alpha.\beta$.
Earley's algorithm differs from the LR parser-generation algorithm in two respects :
- it constructs a new itemset on the reading of a new terminal
- on the reduction of an item it does not place a reduce entry in a table but it calls the routine COMPLETER.
We combine Earley's technique with the LR parser-generation algorithm by treating every LR shift symbol as an Earley input symbol. If the shift symbol is a nonterminal then only the core of the new itemset will be created. We call such an itemset a "bookkeeping itemset" and it will not be treated further. It functions only as a set of all the items which will originate from the shifting or reducing on the nonterminal. This set will be used by the routine COMPLETER. The activation of the routine COMPLETER in Earley's algorithm reads as follows:

```
- if $I_i$ is the current itemset
        - if $<A :: a., f>$ is the item under observation
          then do
                - ( COMPLETER: )
                if a = input symbol
                then for each item $<B::\alpha.A\beta, h>$ in $I_f$ do add $<B::\alpha A.\beta, h>$ to $I_i$.
```

We use this part of the procedure in the function LR-TABLE. Line 2b of it reads as follows :

- if $[A::\alpha.X]$ is in I_i, then add 'reduce p' to ACTION(i, X), where p is production $A::\alpha$.

It has to become :

- if $<A::\alpha.B, f>$ is in I_i, then add the core of GOTO(I_f, A) to GOTO(I_i, B).

In the enhanced LR parser-generation algorithm a father itemset number has to be added to all LR-items in order to get Earley-items. In the algorithm of Earley it is possible that by the addition of $<B::\alpha A.\beta, h>$ to I_i a new reduction has to be performed because β can be empty.
A natural property of the enhanced LR parser-generation algorithm is that GOTO(I_f, A) can not contain completed items.
However, when the sub-formalisms of negation and cooperation are used in the grammar the schema for the creation of GOTO(I_f, A) can not longer be followed because of the interaction with other rules. The sequence of reductions then becomes important. In that case the original Earley strategy has to be followed.
This enhanced LR parser generation algorithm for the creation of a FSA will also automatically handle grammars with left recursion.

If the grammar is middle or right recursive the algorithm will not terminate. This situation will be discussed in (b).

Example for (a).
Suppose we want to construct a FSA for the grammar

A :: A, a | B, b | c .
B :: A, b | B, a | d.

As usual, we add the rule S' :: (A) .

I_1 contains the item :
core: <S' :: .(A), 1> On a shift on '(' we construct I_2.
I_2 contains the items :
core: <S' :: (.A), 1>
closure: <A :: .A a, 2>, <A :: .B b, 2>, <A :: .c, 2>,
<B :: .A b, 2>, <B :: .B a, 2>, <B :: .d, 2>.
On a shift on B we construct I_3. On a shift on 'c' we construct I_5.
On a shift on A we construct I_4. On a shift on 'd' we construct I_6.

I_3 contains the items :
core: <A :: B .b, 2>, <B :: B .a, 2>
No closure has to be added. This is a bookkeeping itemset and will not be processed further.

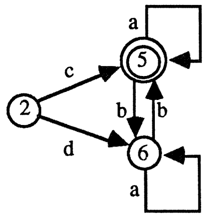
I_4 contains the items :
core: <S' :: (A .), 1>, <A :: A .a, 2>, <B :: A .b, 2>
No closure has to be added. This is a bookkeeping itemset and will not be processed further.

I_5 contains the items :
core: instead of <A :: c., 2> we add GOTO(2, A), which is I_4:
<S' :: (A .), 1>, <A :: A .a, 2>, <B :: A .b, 2>.
No closure has to be added.
On a shift on 'a' we construct the following itemset J_1.
J_1 contains the items :
core: instead of <A :: A a., 2> we add GOTO(2, A), which is I_4:
<S' :: (A .), 1>, <A :: A .a, 2>, <B :: A .b, 2>.
The core of this itemset J_1 is equal to the core of I_5. Therefore, on a shift on 'a' we shift to I_5.
On a shift on 'b' we construct I_6.
On a shift on ')' we accept.

I_6 contains the items :
core: instead of <B :: d., 2> we add GOTO(2, B), which is I_3:
<A :: B .b, 2>, <B :: B .a, 2>
No closure has to be added.
On a shift on 'a' we construct the following itemset J_2.
J_2 contains the items :
core: instead of <B :: B a., 2> we add GOTO(2, B), which is I_3:
<A :: B .b, 2>, <B :: B .a, 2>
The core of this itemset J_2 is equal to the core of I_6. Therefore, on a shift on 'a' we shift to I_6.

On a shift on 'b' we construct the following itemset J_3.
J_3 contains the items :
core: instead of <A :: B b., 2> we add GOTO(2, A), which is I_4:
<S' :: (A .), 1>, <A :: A .a, 2>, <B :: A .b, 2>.
The core of this itemset J_3 is equal to the core of I_5. Therefore, on a shift on 'b' we shift to I_5.

This ends the construction of the itemsets. We only created shift entries in the LR-table which can therefore straightforward be converted into the following FSA (where we identify an LR-itemset which shifts on the endmarker ")" with an accepting state of the FSA):



The FSA recognizes the language which is generated by the grammar
A :: A, a | B, b | c .
B :: A, b | B, a | d.
This can be verified by rewriting the grammar into a regular expression. We do this by first eliminating B. The second rule can be rewritten as
B :: B, a | [A, b | d]1.
Therefore B => [A, b | d]1, [a]*. Substitution in the first rule gives :
A :: A, a | c | A, b, [a]*, b | d, [a]*, b. which is equal to
A :: A, [a | b, [a]*, b]1 | [c | d, [a]*, b]1. Therefore
A => [c | d, [a]*, b]1, [a | b, [a]*, b]* , which is expressed by the FSA above.

(b). It is well known that for regular expressions and for regular (type-3) grammars a deterministic FSA can be constructed. A regular expression may also be written as a cfg with left or right recursion. It may therefore be expected that for cfg's with only left and/or right recursion a FSA might be constructed. In fact, this is suggested by the transformation of tail recursion into iteration, but we are not aware of an algorithm for the direct generation of a FSA for such a cfg. In the compiler we provide for the construction of a FSA for cfg's with left and/or right recursion by an extension of the technique which we explained in (a). The extension is motivated by the following observations.
If in itemset I_i the two Earley-items <A::$\alpha_1.\beta$, f_1> and <B::$\alpha_2.\beta$, f_2> are present, then we know that both items will travel along any sequence of terminals which can be generated by β. We call such items "compatible". If we are not interested in the parsing of the grammar but only in its recognition then it seems unnecessary to let both items travel. We can choose one of them as a representative, until we reach the itemset where the item will reduce. At that point we have to perform the action of the COMPLETER which says that the core's of the itemsets GOTO(I_{f1}, A) and GOTO(I_{f2}, B) have to be added to the itemset which is to be constructed. Suppose that these core's contain mutually compatible Earley-items. Then again it is only necessary to add one of these core's. In such a way, by comparing items for the languages that they are able to generate, it becomes possible to discard items which will behave in the future always in the same way as other items.

We generalize this idea in the following definition and extensions to the LR parser-generation algorithm.

Definitions.
We call the two items $<A_1::\alpha_1.\beta_1, F_1>$ and $<A_2::\alpha_2.\beta_2, F_2>$ compatible if
- β_1 and β_2 generate the same language and
- the sets of fathers F_1 and F_2 are compatible.

We call two sets of fathers F_1 and F_2 compatible if
- $F_1 = F_2$ or
- for each $f_1 \in F_1$ there exists a $f_2 \in F_2$ and for each $f_2 \in F_2$ there exists a $f_1 \in F_1$ such that the core's of the itemsets GOTO(f_1, A_1) and GOTO(f_2, A_2) are compatible.

We call the core's of two itemsets I_i and I_j compatible if
- core of I_i = core of I_j or
- for each item $i_n \in I_i$ there exists an item $i_m \in I_j$ and for each item $i_m \in I_j$ there exists an item $i_n \in I_i$ such that i_n and i_m are compatible.

Δ

Extension to the LR parser-generation algorithm.
An Earley-item will not be added to an itemset if this itemset contains already a compatible Earley-item.
A newly created itemset will not be added to the set of constructed itemsets if it is compatible with an already existing itemset.

Implementation in the compiler.
For practical reasons we compare in the compiler only items of the form $<A::\alpha.\beta, f_1>$ and $<A::\alpha.\beta, f_2>$, that is, these items differ only in the father itemsetnumber.
Compatibility between items and itemsets will only be calculated if necessary. The outcome of the calculation is stored in a datastructure.
The definition of compatibility is recursive. It is possible that during the calculation of the compatibility of two items that same compatibility is called for. We resolve the circularity by the observation that, if the outcome of a compatibility calculation depends solely on the outcome of the calculation itself, we are free to choose true or false for the outcome. In that case we will choose the value true. We will clarify this by an example.

Example for (b).
Our example concerns a grammar which is believed by Earley (1970) to be one of the few grammars for which his parsing algorithm takes cubic time (see also the discussion on worst-case languages and grammars in chapter 7). During parsing an exponential number of parse trees will be created. The grammar, which is called in the literature "UBDA", is:

S :: S, S ; a.

We will show how the enhanced LR construction algorithm (according to a. and b.) creates for this grammar a FSA with 2 states.
As usual, we add the rule S' :: (S) .

I_1 contains the item :
core: <S' :: .(S), 1>
On a shift on '(' we construct I_2.

I_2 contains the items :
core: <S' :: (.S), 1>
closure: <S :: .S S, 2>, <S :: .a, 2>.

On a shift on S we construct I_3. On a shift on 'a' we construct I_4.

I_3 contains the items :
core: <S' :: (S .), 1>, <S :: S .S, 2>
No closure has to be added. This is a bookkeeping itemset and will not be processed further.

I_4 contains the items :
core: instead of <S :: a., 2> we add GOTO(2, S), which is I_3:
<S' :: (S .), 1>, <S :: S .S, 2>
closure:<S :: .S S, 4>, <S :: .a, 4>
On a shift on S we construct I_5. On a shift on 'a' we construct I_6.

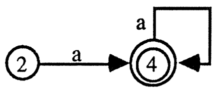
I_5 contains the items :
core: 3 items will shift or reduce on S:
1. instead of <S :: S S., 2> we add GOTO(2, S), which is I_3:
<S' :: (S .), 1>, <S :: S .S, 2>
2. normal shift of <S :: S .S, 4>
3. instead of <S :: a., 4> we add GOTO(4, S), which is I_5. This is a tautology.

Now the extension of the algorithm according to b. comes into action. I_5 contains <S :: S .S, {2}> and <S :: S .S, {4}>. We have to question if these two items are compatible. That will be so if GOTO(2, S), which is I_3, is compatible with GOTO(4, S), which is I_5. Therefore the question is : are I_3 and I_5 compatible ?
Both contain the item <S' :: (S .), 1>. The remaining item in I_3 is <S :: S .S, 2> and in I_5 the remaining items are <S :: S .S, {2, 4}>. The question therefore is if the items <S :: S .S, 2> and <S :: S .S, 4> are compatible. But that is the question which we started with. This is again a tautology : the answer will solely depend on the answer. We choose therefore "true" and store this decision in a table. In itemset I_5 we will now leave out item <S :: S .S, 2> because its future will be included in the future of the item <S :: S .S, 4>. We recapitulate :

I_5 contains the items :
core: <S' :: (S .), 1>, <S :: S .S, 4>
No closure has to be added. This is a bookkeeping itemset and will not be processed further.
After the creation of an itemset we have to compare it with already existing itemsets. I_5 is compatible with I_3, as we saw already. Instead of I_5 we will therefore take I_3.

I_6 contains the items :
core: instead of <S :: a., 4> we add GOTO(4, S), which is I_5:
<S' :: (S .), 1>, <S :: S .S, 4>
The core of this itemset is compatible with the core of I_4. Instead of I_6 we will therefore take I_4.

We are now finished with the creation of itemsets. The constructed FSA is :



6.4.2 Optimization of the treatment of empty- and unit -rhs's

The usual approach for the treatment of empty rules is to introduce the empty symbol ε and to perform for each created itemset A an ε-move. The resulting itemset B has to be merged with A and the empty items have to be removed. This process may be repeated until no more empty items in A are present. A more efficient approach is to make shortcuts in the datastructure for grammar rules over nonterminals which have an empty rhs. But then care has to be taken of the correct representation of ε-reduces in the parse, as well as of the intermediate actions.
The usual approach to the treatment of unit rules is to simulate the runtime treatment of these unit reduces in compile-time. Then the intermediate actions have to be sampled into the extended LR-table (we give an example in the next section).
The treatment of both kinds of rules is more complicated when the sub-formalisms of cooperation and negation are used. The treatment of cooperation asks for the removal of items for which the cooperation with other items in the itemset is not fulfilled. Markers of negation dots may fire other items with dots with that marker, which then have to be removed. Special care has to be given to the removal of starting items with a cooperation : it is possible that their fatheritems in the same itemset also have to be removed.
The optimization for left- and/or right recursive markers, together with the compatibility checking of items and itemsets, still further complicates the treatment. However, when the usual approach is followed which we sketched above, and when the normal treatment during runtime is simulated properly in compile time, the implementation is not complicated. However, the speed of compilation decreases. In our implementation we tried to improve the speed by doing more preprocessing in the module "Preparesets". The results will be published elsewhere by M. Elstrodt and the author.

6.4.3 Disambiguation of shift-reduce/reduce conflicts

Normally, the PTA takes care of all indeterminism which is caused by ambiguities in a grammar. However, the user of the grammar may restrict the number of parses for an ambiguous grammar by asking for a global arbitration in the case of conflicts in the LR-table for an itemset. This arbitration is straightforward to implement because LR-parsing tries to follow all derivations in parallel. In the current implementation we provide for the following 3 possibilities, which are indicated by the setting of switches:
- - "shift_no_reduce" : in case of a shift/reduce conflict in the LR-table: precedence of shift
- - "reduce_no_shift" : in case of a shift/reduce conflict in the LR-table: precedence of (all) reduce(s)
- - "one_cs_reduce" : in case of a reduce/reduce conflict between two or more rules : precedence of a rule with a longer rhs; if the length is the same : the rule which was placed first in the grammar.

6.4.4 The principle of the datastructure for the extended LR-table

Output of the module GenLRsets will be an extended LR-table from which the code for the PTA can be generated. The LR-table will be constructed in a datastructure which will allow for the generation of code for the PTA in a straightforward manner.
In section 4.2.2 we specified the syntax of the programs which have to be generated. This syntax specifies a sequence of operations. The sequence of operations within actions which is specified within a grammar rule has to be found back in the sequence of generated instructions. However, the sequence of grammar rules within a grammar is not important and does

not have to be reflected in the generated code. As we discussed in section 4.5 this allows for the processing on parallel hardware.
The parallelism is caused by the non-determinism which is found in the constructed LR-table when multiple entries are allowed for the action on a symbol. In fact, for the action on a symbol zero or one shift may be present, zero or one accept and zero or more reduces may be present. The entries become more complicated when the optimization for empty and unit reduces is taken into account or when code has to be generated for nonterminals which are rewritten with cf rules which do not contain infix recursion (the switch multi=false, which optimization was discussed in section 6.4.1). Then it is possible that, before the execution of the instruction LSTK for a shift or the instruction RDCF for a reduce, a chain of intermediate reductions has to be performed by the instruction PLRDC. For instance, consider the following grammar in which we abbreviate actions by roman numbers.

(1) a, B :: B, a .
(2) B :: C {I}.
(3) B :: D .
(4) C :: e {II}, [f] .
(5) D :: [e] {III}, [g].

Suppose the first itemset

(1) a, B :: $_1$B $_2$a
(2) B :: $_4$C {I}
(3) B :: $_6$D
(4) C :: $_8$e {II} [f]
(5) D :: [$_{11}$e] {III} [$_{12}$g]

is constructed from which we want to create action entries on the symbol "e". Initially, we create a shift entry to the successor state for item pairs (8, 9) and (11, 12) and a reduce entry for the items 8 and 11. However, if we optimize for unit reductions further reductions may be indicated for the items 4 and 6. In their turn, both unit reductions give rise to a shift for item pair (1, 2). Therefore, the action on symbol "e" results in a shift-entry in the LR-table, with a number of intermediate reduces. These reduces we called in section 4.2.2 "pseudo-reduces" and are performed by the instruction PLRDC. If the switch intermediate_unit_reduction is false then the user is not interested in the representation of the unit reductions. In that case we can leave them out. (If the switch build_parse is false the user is not interested at all in the representation of reduces.)
The order in which the pseudo-reduces have to be performed has to be represented in the datastructure for the LR-table, but also the parallelism of the reductions from the items 4 and 6. Furthermore, the references to the actions I, II and III have to be kept.

From section 4.2.2.3.1 we repeat the essential syntax rules for the piece of a program which corresponds with an entry in a LR-table. The datastructure for the LR-table closely resembles this syntax.

```
code_per_symbol       :: [ LSTK , statenr_1 ] , [ list_of_instructions ]*, EXT |
                         NSTK , statenr_1 , EXT.
list_of_instructions  :: TOP , from_goto_action , [ instruction ] , [ list_of_instructions ]* .
from_goto_action      :: item_current , item_projective, [ action_call ]* .
instruction           :: PLRDC , leftsymbol_cf  |
                         LRDCF  , leftsymbol_cf  |
                    LRDCS  , statenr_r  |
                    ACC .
```

We comment on each line for those aspects which concern the datastructure of the LR-table.

```
- code_per_symbol:   - there is a reserved position for the shift itemset number "statenr"
                       which will be filled in as soon as that number is known
                     - a parallel accept and/or zero or more parallel reductions are stored in
                       a structure which corresponds with "list"
- list_of_instructions: - in from_goto_action the itemnumbers and references to actions
                       which are associated with a shift or a reduce are represented
                     - instruction is one of the 3 instructions RDCF, PLRDC or LRDCS
                       for a reduce or an accept (or no instruction when there is only a
                       shift)
                     - after the instruction other reduction instructions may follow, each
                       represented by a "list"
- from_goto_action : - item_current, item_projective, zero or more actions
```

Therefore, the essential datastructure for the action on a symbol in the LR-table is of the form

```
    [shift_statenr], [item_current, item_projective, [ref_to action]*,
                     [reduce instruction], [reduce_list]* ]* .
```

For the itemset of our example the entry in the LR-table for symbol "e" will become (where open brackets on different lines in the same column indicate the start of parallel actions) :

- when no optimizations are involved (switch multi=true) :

```
    successorstate, (8, 9, II)
                    (8, 10, II, RDCF C)
                    (11, 12, III )
                    (11, 13, III, RDCF D)
```

- when the optimization for as much FSA code (switch multi=false) and for unit reductions is performed:

```
    successorstate, (8, 9, II)
                    (8, 10, II, PLRDC C, (4, 5, I, PLRDC B, (1, 2) ) )
                                         (6, 7,  PLRDC B, (1, 2) )
                    (11, 12, III )
```

- when also the optimization for compatibility checking is performed (switch test_comp_sets=true):

```
    successorstate, (8, 9, II)
                    (8, 10, II, PLRDC C, (4, 5, I, PLRDC B, (1, 2) ) )
                    (11, 12, III )
```

- when also intermediate nonterminals in unit reductions have to be left out (switch intermediate_unit_reduction=false) :

```
    successorstate, (8, 9, II)
                    (8, 10, II,  (4, 5, I, PLRDC B, (1, 2) ) )
                    (11, 12, III )
```

- when also the user is only interested in recognition and not in parsing :

```
        successorstate, (8, 9, II,  (4, 5, I, (1, 2) ) )  )
                        (11, 12, III )
```

- and also, finally, when the actions I, II and III were not present in the grammar :

```
        successorstate.
```

6.4.5 The algorithm for the construction of itemsets

The original algorithm of Construct-itemsets, as a combination and extension of the algorithms "Closure", "Goto-itemset" and "LR-table", was specified in section 5.0 .

The new algorithm, which we call GenLRsets, makes use of a queue of created itemsets for which an LR-table has to be created. It makes also use of the datastructures which were constructed in the module "Preparesets" and which were listed in section 6.3 .
The algorithm may be invoked for all rhs's together or for each lhs separately. It reads as follows.

Algorithm GenLRsets

```
{initialize }
- if algorithm is invoked for a lhs
  - Itemset[1] := first item of lhs
- else
  - if switch transduction = true
    - Itemset[1] := start_cs_items
  - else
    - Itemset[1] := {[S' :: .(, S, )] }
- #created_itemsets := 1
- put a reference to Itemset[1] in the queue

  { treat next itemset in the queue }
- while the queue is not empty do
  - take the reference to the next itemset from the queue and call the itemset: Current_itemset

    { prepare treatment of don't cares and s : because shifts on don't cares and lines are
        independent of the symbol they are treated separately; determine the goto-items on
        these symbols }
  - Temp := Current_itemset $\cap$ items_before_notion[dontcare_symbol]
  - Items_projective_of_dontcare_items := $\emptyset$
  - for each item i in Temp do
    - Items_projective_of_dontcare_items := Items_projective_of_dontcare_items $\cup$
        follow_items_of_item[i]
  - Line_items := Current_itemset $\cap$ items_before_notion[line_symbol]
  - Items_projective_of_line_items := $\emptyset$
  - for each item i in Line_items do
    - Items_projective_of_line_items := Items_projective_of_line_items $\cup$
        follow_items_of_item[i]

    { create new itemsets }
  - for each symbol X, except for the don't care symbol and the line symbol, do
- New_itemset := ∅ { in New_itemset the new itemset will be created }
- Temp := Current_itemset ∩ items_before_notion[X]
- for each item i in Temp do
   - New_itemset := New_itemset ∪ follow_items_of_item[i]
    { add the line-items of Current_itemset }
   - New_itemset := New_itemset ∪ Line_items
    { add the goto-items of the dontcare-items }
   - if X is a terminal
      - New_itemset := New_itemset ∪ Items_projective_of_dontcare_items
    { add the goto-items of the line-items with X behind the line-symbol }
- Temp := Items_projective_of_line_items  ∩ items_before_notion[X]
- for each item i in Temp do
   - New_itemset := New_itemset ∪ follow_items_of_item[i]

- if user switches indicate : disambiguate in case of shift/reduce-reduce conflicts

    { Place reduce-entries in LR-table for the current itemset}
- Cf-reduce_items := New_itemset  ∩ all_cf_reduce_items
- Cs-reduce_items := New_itemset  ∩ all_cs_reduce_items
- if switch multi=false (the optimization for grammars without
            middle-recursion has to be performed)
   - call the enhanced COMPLETER routine of section 6.4.1 for
            Cf-reduce_items
- else
   - remove items from Cf- and Cs-reduce_items with unfulfilled
                  cooperations
   - for each item i in Cf-reduce_items do
      - create a cf-reduce-entry in the LR-table for (Current_itemset, X)
                  together with triple (item_current, item_projective, actions)
   - for each item i in Cs-reduce_items do
      - create a cs-reduce-entry in the LR-table for (Current_itemset, X)
                  together with triple (item_current, item_projective, actions)
     { remove reduce-items from New_itemset }
- New_itemset := New_itemset  ∩ not Cf-reduce_items  ∩ not Cs-reduce_items

- remove items from New_itemset with unfulfilled cooperations

    { determine if core of new itemset  already exists; a hashcoding is used for
   the core of itemsets which is stored in the array
            Hashcode_core_of_itemsets[1..#itemsets] }
- Hashcode_core_of_newset := a hashcoding of the datastructure for the core of
            New_itemset
- New_itemset_exists := (there is a j such that j ∈ Hashcode_core_of_itemsets
            [Hashcode_core_of_newset]) and (New_itemset = Itemset[j])
- if New_itemset_exists = false
     { if algorithm is invoked for rhs's: add closure items to new itemset}
    - if New_itemset <> ∅
       - if switch transduction = true
          - New_itemset := New_itemset ∪ start_cs_items
       - Items_in_newset_before_nonterminal := New_itemset  ∩
             nonterminals_after_start_or_middle_items
            - for each item i in Items_in_newset_before_nonterminal do
                - New_itemset := New_itemset ∪
                    closure_items_of_item[notion_behind_item[i]]
        - #created itemsets := #created itemsets+1
        - Itemset[#created itemsets] := New_itemset
        - put a reference to New_itemset in the queue
        - shift_setnr := #created_itemsets
    - else
        - shift_setnr := j

    { Place shift-entry in LR-table for current itemset}
    - if X = endmarker
        - create in the LR-table for (Current_itemset, X) an accept-entry
    - else
        - create in the LR-table for (Current_itemset, X) a shift-entry to
          shift_setnr
{ end of  GenLRsets }
```

6.5 Generating code for a PTA : modules "Supscanner" and "SupLRcode"

The modules "Supscanner" and "SupLRcode" take as input the datastructure for the LR-table which we discussed above. We already indicated that this datastructure resembles closely the sequence of the code which has to be generated.
In the module "Supscanner" the scanner-table is constructed which we described in section 4.2.2.3.1 . The module "SupLRcode" generates the code for L- and R-states. The syntax of the code for L-states was described in the same section. The syntax for R-states could have been the same, but because the formalism of a lhs is more restricted then the formalism of a rhs the syntax for the code of a R-state can be more limited. It was described in section 4.2.2.3.2 .
In chapter 4 we described the working of the different instructions. In section 6.5 we described the relation between the extended LR-table and the code which has to be generated.
There remain a number of topics which may be of interest to be described here.

6.5.1 Shared code

Code is shared as much as possible. Candidates for sharing are :
- code for actions; this code is generated already in the module "Readgrammar" and is written in a separate place
- code for reductions.

As an example we present the generated code for the entry in the LR-table which we discussed in section 6.4.4. The itemset was

(1) a, B :: ${}_1$B ${}_2$a
(2) B :: ${}_4$C {I}
(3) B :: ${}_6$D
(4) C :: ${}_8$e {II} [f]
(5) D :: [${}_{11}$e] {III} [${}_{12}$g]

The entry in the extended LR-table was, when the optimization for as much FSA code (multi=false) and for unit reductions is performed:

```
        successorstate, (8, 9, II)
                        (8, 10, II, PLRDC C, (4, 5, I, PLRDC B, (1, 2) ) )
```

```
                                    (6, 7,  PLRDC B, (1, 2) )
                    (11, 12, III ).
```

We display the generated code as it is created by our disassembler. Successorstate=3 and the start of the code for the entry is at program counter=67 :

```
21 -16 -15       PLRDC     Pseudo CF-reduce, reduced symbol: B
23 -17  -1   2   CTOP (    item_current: -1, item_projective: 2
26 -18           TCL  )
27 -20           RTN

28 -16 -16       PLRDC     Pseudo CF-reduce, reduced symbol: C
30 -17  -4   5   CTOP (    item_current: -4, item_projective: 5
33 -21   1       ACAL      Call action-subroutine 1
35 -19  21       CAL       Call subroutine 21
37 -18           TCL  )
38 -20           RTN

67  -2   3       LSTK      Shift to state: 3
69 -17  -8   9   CTOP (    item_current: -8, item_projective: 9
72 -21   4       ACAL      Call action-subroutine 4
74 -18           TCL  )
75 -17  -8  10   CTOP (    item_current: -8, item_projective: 10
78 -21   4       ACAL      Call action-subroutine 4
80 -19  28       CAL       Call subroutine 28
82 -18           TCL  )
83 -17 -11  12   CTOP (    item_current: -11, item_projective: 12
86 -21   7       ACAL      Call action-subroutine 7
88 -18           TCL  )
89 -13           EXT       Exit from the code for this symbol
```

6.5.2 Skip instruction with trees

If an itemset expects only the 2 symbols "a" and "." (the rest symbol), and on the "." it shifts to itself, then for recognition purposes all characters in the input may be skipped up to the "a". We use this observation for the generation of a skip instruction for closing tree-brackets. Especially in the case of large tree-structured textfiles this is a useful instruction. With each open tree-bracket a pointer may be maintained to the corresponding closing bracket.

6.5.3 Variables

The generation of code for expressions of variables is straightforward. In runtime a stack is maintained during evaluation, and all operations concern the 2 topmost elements of the stack or a variable and the topmost element. The instructions for operations are :

PSH variable_name:	push variable or literal onto the stack
CAT variable_name:	concatenate a variable with topmost element; replace top by resulting element
ASS variable_name:	assign topmost element to variable
TEQ:	test equality of 2 topmost elements
TNE:	test inequality of 2 topmost elements

The transport of variables to closure items and from reduce items is governed by the following instructions :

ALL integer:	# variables in the grammar_rules
RCV #variables variables:	receive the value of a variable from a symbol_infolist in the PTA
SND #variables variables:	send the value of a variable to the formal parameter of a closure item
CLI integer:	create a starting item with this number.

Variables are declared as Input ("I") and/or Output ("O") parameters. We give the rules for the construction of code for the passing of variables.
Variables are local to the grammar rule where they appear. We number them for each rule as an integer, starting with 1.
- Suppose the rule A :: a $_{13}$B(x, y) $_{14}$b , where x and y are O-variables; the internal numbers of x and y are 3 and 4. On a shift on B from 13 to 14 the following code has to be generated:

```
RCV 2
3 4.
```

This says that the 2 variables which are present in the symbol-infolist (in runtime in the PTA) which belongs to the reduced symbol B have to be received by variables 3 and 4.
- Suppose a grammar which contains, among others, the rules (in which we indicate some itemnrs) :

$A :: \alpha\ _{14}a\ _{15}B(x, y)\ \beta$
$B_1(I{:}n_1, I{:}m_1) :: {}_{-20}\ a_1\ _{21}\gamma_1$
$B_2(I{:}n_2, I{:}m_2) :: {}_{-25}\ a_2\ \gamma_2$ (starting items always have a negative number).

Suppose the local numbers of x and y are resp. 3 and 4.
Then the item 15: $A :: \alpha\ a\ _{15}B(x, y)\ \beta$
gives rise to the following closure-items :

$B_1(I{:}n_1, I{:}m_1) :: {}_{-20}\ a_1\ \gamma_1$
$B_2(I{:}n_2, I{:}m_2) :: {}_{-25}\ a_2\ \gamma_2$

On the generation of code for an itemset with item 14 which will shift on an "a" to an itemset which contains item 15 we generate:

```
TOP 14 15          action-subroutine A:  ALL 2
CLI -20                                  SND 2
ACAL A                                   3 4
CLI -25                                  RTN
ACAL A
...
TCL
```

Normally, a starting item is not represented in a runstate. With the instruction CLI it is created. The instruction ALL creates 2 variables in the infolist of the current item; these are filled in by the instruction SND which has 2 arguments : the variables 3 and 4 which belong to item 14.
- Suppose in the example above the starting item -20 shifts to its goto-item(s), for instance 21. Then the following code is generated :

```
TOP -20 21
ALL N              where N is the number of local variables in the
```
rule $B_1(I{:}n_1, I{:}m_1) ::\ a_1\ \gamma_1$.

6.5.4 Lexicon operations

We summarize from chapter 4 the working of the instructions LEXST, LEXINC and LEXRDC. LEXST initialises the reading in the lexicon by putting the lexicon-pointer in an infolist at the start of the lexicon. After each character that is read from the input a transition is made in the lexicon by the instruction LEXINC and a check is made whether the end of an entry is reached. In that case the instruction LEXRDC creates a connector and returns in variables the categories which are present with the entry. The transition in the lexicon and the creation of a connector resemble the normal instructions LSTK and LRDCF which are generated on the shift and the reduce of an item. This brings us to the formulation of a lexicon symbol as a regular expression which generates an indefinite number of terminal symbols :

$LEX :: [*]+ .

The following simple tricks allows us to profit from the existing algorithm GenLRsets :

- add implicitly to a grammar the rule $LEX :: [*]+ and remember the item number of $LEX :: .[*]+ in i1 and that of $LEX :: [*]+. in i2
- add to an item α .$X, where $X is an arbitrary lexicon symbol, as a closure the item $LEX :: .[*]+
- make some minor extensions to GenLRsets such that the following code will be created on the shift of i1 to i2:

```
        if i1 is a closure item  if i1 is not a closure item   (this is detected by the
                                                               bookkeeping for regular
                                                               expressions)
        CTOP -i1     i1          TOP   i1      i1
        CAL   r1                 CAL   r2
        CTOP -i1     i2          TOP   i1      i2
        CAL   r1                 CAL   r2
        LEXRED                   LEXRED        (lexicon reduce with nonterminal = first
                                                 category behind the entry)

r1:     LEXST                    (put lexicon pointer "place" at the start of the lexicon)
        RTN

r2:     LEXINC                   (shift lexicon pointer)
        RTN
```

The subroutines at r1 and r2 are generated at the start of the compiler.

6.6 Cascaded grammars : modules "Linking loader" and "Disassembler"

By the implementers of the system Parspat a linking loader and a disassembler were developed. The linking loader serves to synchronize the internal numbering of symbols within cascaded grammars. It accepts as input a number of individual grammars, the symbol-tables for these grammars and a file which describes the sequence in which they have to be placed. It outputs a program file in which the symbols are renumbered.
The disassembler serves to make the generated code readable.

6.7 Relevance of the compiler for other purposes

The compiler can be used for a number of other purposes which are not directly related to the compilation of code.

The module Readgrammar can be used separately in order to create a datastructure for a grammar, together with a symbol table, for any purpose. It can also be used to build on-line a FSA for regular expressions.
The module GenLRsets can directly be used as an Earley parser when the switch "multi" is set to false and the switch "interactive" to true. All sub-formalism of the unifying formalism can then be used, except type-1 and -0 rules.
The optimizations which are built into the compiler allow for the creation of a FSA for subsets of cfg's together with lines, arbs, don't cares and negations. The generated code can be transformed into code for other systems which simulate the behaviour of a FSA.

6.8 Possible improvements of the compiler

There are a number of possibilities for the improvement of the compiler. Among them we name :
- to let the modules work not sequentially in time, but as cooperating processes
- to compile "lazy", that is to create only an itemset when the runsystem asks for it.

6.9 The choice of the programming language

For the implementation of the system Parspat we chose the programming language Pascal because of its widespread availability, its readability and its possibility to generate efficient programs on small and large machines.
Within a Dutch working group on the use of Pascal in the Humanities a string- and set-package was developed, written in standard Pascal (Elstrodt, Honig, Masereeuw, Portier, Schwartzenberg, Skolnik, Van der Steen and Van Halteren, 1984). The characteristics are :
- efficient storage of variable length strings and sets
- efficient implementation of a large number of string and set operations
- standard calls for direct access i/o
- in benchmarks proven to be portable over a range of different machines.

The implementation of Parspat makes extensive use of this package.

7. Complexity

In this chapter we want to investigate the complexity of the compiler and the runtime system, as designed in chapters 4, 5, and 6. We will determine the complexity for each of the subformalisms and will relate it to the complexity of already existing strategies for these subformalisms which are known from the literature and which were listed in section 2.6 . The discussion will be structured as follows.
For each of the subformalisms we will determine the time and space complexity, for the algorithms within the compiler as well as within the runtime system. In the comparison with other strategies a number of possibilities arise : (a) the complexity of our algorithms may be better, (b) equal or (c) worse than the other ones. In the latter case we will include a better algorithm for the subformalism in question, stated as an extension to the algorithms which we already developed. For instance, we will show how the compiler can be improved in order to obtain a sub-linear time-complexity for cfg's which describe regular languages. In order to explore the extensibility of the algorithms within the system Parspat we will indicate also, if possible, how in our opinion further improvements in complexity can be made. Sometimes we will discuss briefly other applications of our algorithms.
The discussion concerns formal languages, grammars and automata which are related to each other by the process of parser generation. We will smooth out some of the inconsistencies in the complexity of recognizers which are automatically constructed for a grammar and those which are constructed for the language which is described by that grammar.
The power of the PTA is that it works online and that the exponential character of backtracking is avoided. Because of the richness of the unifying formalism this may have an implication for other formalisms which still suffer from implementations with an exponential behaviour, like Prolog and knowledge-based systems. Our approach is to store only once a data-element which will, by its interpretation, give rise to subsequent processing. This principle is found in the compiler, where identical items and itemsets are identified, and in the PTA where identical dagnodes and data-elements to which they refer, like variables, are identified. By avoiding the exponential character of backtracking other exponential traits become more visible which are of interest when one tries to get a deeper insight in the problem of NP-completeness. We will investigate the NP-problem of the parsing of linear context sensitive grammars and will suggest an improvement of the PTA in order to remove another origin of exponential behaviour.

7.1 The complexity of compilation

In chapter 6 we discussed the different modules of the compiler. The complexity of the compiler as a whole is determined by the complexity of each of these modules. We will investigate them separately.

Readgrammar.
The algorithm of Readgrammar constructs a NFA online for each side of a grammar-rule. On the appearance of any symbol in the input a number of elementary operations is executed

which is bounded by the number of alternatives in a lhs or rhs. We conclude that the time and space complexity are proportional to the size of the grammar.

Preparesets.

In the module Preparesets a number of sets is created. Some of them need a depth-first walk through the datastructure for the grammar rules. The walk always starts on a notion-record and ends on a notion-record, with a visit to each intermediate goto-record. With a notion-record an item is related. With a goto-record is related the start of an alternative or the start of a regular expression. Therefore, the calculation of a relation between two items costs a number of elementary operations which depends on the number of brackets in the regular expressions in a side of a rule. If the total number of lhs's and rhs's in the grammar is s, the longest side has length l and the maximum number of brackets in a side is b then the construction of the array follow_items_of_item will cost at most $(s.l)^2.(b+1)$ time . The construction of all other sets will cost less.

GenLRsets.

The algorithm GenLRsets constructs from a Current_itemset for each expected symbol a New_itemset. We will discuss the complexity of this process for a number of subformalisms. The complexity of compilation is especially important in the case of pattern matching. In other subsections of this chapter we will compare our runtime behaviour with other existing pattern matching methods. All these methods have a linear preprocessing time. We choose therefore as subformalisms :

I. grammars which contain regular expressions
II. type-4 grammars with lines, arbs and don't cares
III. type-4 grammars with negations and don't cares.

I. grammars which contain regular expressions

A bound for the complexity of the subformalism of regular expressions may be found by the observation of the worstcase grammar

$$S :: [a \mid b]^*, a, [a \mid b]^{n-1}$$

which generates any string of a's and b's in which the n'th character from the right is an "a". Aho, Sethi and Ullman (1986, p. 128) observe that for this grammar no DFA may be constructed with fewer than 2^n states because at least 2^n states are required to keep track of all possible sequences of n a's and b's. The algorithm GenLRsets (together with the construction of sets in the module Preparesets) transforms the NFA which is constructed by Readgrammar into a DFA. We may therefore conclude that the worstcase behaviour of GenLRsets is exponential, in time and in space, and that no improvement may be expected. The calculation of compatibility is, in the worstcase, quadratic in the number of itemsets which are generated. This is caused by the fact that in the recursive definition of compatibility the mutual compatibility of 2 itemsets is asked for. In the worstcase this compatibility has to be calculated for all pairs of itemsets.

II. type-4 pattern-grammars

Little can be found in the literature concerning the complexity of state-generation for a FSA. There seems to be some reservation about the use of finite-state techniques for the creation of pattern matching machines because of the general exponential upper bound on the numbers

of states required for regular expressions. Apostolico and Giancarlo (1986) write about the considerations in the design of new pattern matching algorithms: "As pointed out in (Knuth, Morris and Pratt, 1977), the analysis of the Boyer&Moore (BM) procedure is not simple. This is due to the fact that, when the BM algorithm shifts the pattern to the right, it does not retain any information about characters already matched. Based on this observation, Knuth, Morris and Pratt suggested that the algorithm be made less oblivious by arranging the various situations that could arise in the course of the pattern matching process into a suitable table of "states". Problem is that the number of "states" in such a generalization of the BM strategy can be quite large (the obvious upper bound is 2m, but it is not known how tight a bound this is). Thus the work involved in preparing that table is prohibitive in practice. There is room to suspect that a good portion of the tables is not needed in general. (end quote)".
It seems therefore appropriate to perform some measurements on the time and space which is necessary in the algorithm GenLRsets for the ordinary problem of matching a number of keywords which do not contain regular expressions.
At our disposal are
- the compiler of the system Parspat with which we register:
  - the number of created itemsets
  - the total CPU-time used by the compiler on a MV/4000 computer (0.6 Mips)
- a program which generates a grammar according to the following parameters

- p = 0	no pattern matching but only recognition of a type-4 grammar (in order to investigate the difference with p = 1)
= 1	each rule starts with a line ("-") which effects the pattern matching
- t = 2..9	the size of the alphabet for the terminals of the grammar (there are only terminals); the program selects at random a terminal when a notion has to be written; a check is made such that each generated rule has no duplicate
- k = 2..5	the number of keywords (each keyword is written as a separate grammar rule)
- l = 2..5	the length of a keyword (all keywords have the same length).

Two examples : 1. the parameters p=0, t=3, k=2, l=3 give rise to a grammar
S :: a, c, b.
S :: b, a, a.
2. the parameters p=1, t=2, k=3, l=4 give rise to a grammar
S :: -, a, b, a, a.
S :: -, b, a, b, b.
S :: -, b, a, a, a.

We measure the influence of the variation of each of the parameters p, t, k, and l while keeping the other parameters constant and repeat that measurement a number of times in order to get an impression of the variation of the results. The results are displayed as pairs (number of itemsets, cpu-time in seconds).

Pattern grammar: p.

p =	0		1	
t = 3, k = 3, l = 3	8	4.0	8	6.0
	9	4.0	10	6.8
	8	4.0	10	6.8

t = 3, k = 3, l = 4	11	4.2	13	7.9
	12	4.3	12	7.3
	12	4.3	12	7.3
t = 3, k = 3, l = 5	15	4.6	15	8.4
	11	4.3	16	9.0
	15	4.6	16	9.0

Conclusion : the difference between p=0 and p=1 has almost no influence on the number of generated itemsets, but the cpu-time increases due to the fact that more attempts are made to create itemsets which are equal to existing ones; the numbers increase more or less linear with the length of the keywords.

Variation of the size of the alphabet : t .

t =	2		3		4		5	
p = 1, k = 3, l = 3	8	5.9	9	6.2	10	6.8	10	6.8
	10	6.8	8	5.6	8	5.9	10	6.8
	9	6.3	9	6.2	9	6.2	9	6.3
p = 1, k = 4, l = 3	11	8.5	11	8.3	11	8.4	12	9.3
	11	8.5	11	8.4	10	8.0	11	8.3
	10	7.6	9	7.2	8	6.9	9	7.3

t =	2		3		4		5	
p = 1, k = 3, l = 3	8	9.8	10	11.3	10	11.5	10	11.6
	8	9.8	9	10.4	9	10.5	8	10.2
	9	10.2	9	10.4	10	11.5	8	10.1
p = 1, k = 3, l = 4	10	11.0	11	8.3	12	12.2	12	12.4
	10	11.1	10	12.6	12	12.3	10	11.2
	12	11.8	10	13.2	10	11.3	13	13.5
p = 1, k = 3, l = 5	15	13.5	xx	xx	15	13.8	16	15.4
	15	13.4	xx	xx	15	14.0	16	15.5
	14	13.0	xx	xx	14	13.5	16	15.4

t =	6		7		8		9	
p = 1, k = 3, l = 3	9	10.5	10	11.6	10	11.9	9	10.8
	9	10.7	9	10.7	10	11.7	9	10.6
	10	11.6	10	11.4	9	10.7	10	11.5
p = 1, k = 3, l = 4	13	13.7	13	13.4	13	14.0	13	13.9
	13	13.7	13	13.5	13	13.8	13	13.8
	13	13.7	13	13.7	12	12.4	13	13.8
p = 1, k = 3, l = 5	16	15.2	15	14.3	15	14.1	16	15.9
	15	14.3	16	15.6	15	14.3	16	15.8
	16	15.3	16	15.5	16	15.8	16	15.9

Conclusion : there is almost no influence of the size of the alphabet.

Variation of the number of keywords : k .

k =	2		3		4		5	
p = 1, t = 3, l = 3	8	5.0	9	6.3	10	7.9	11	9.7
	8	5.1	7	5.5	10	7.7	13	11.1
	6	4.4	10	6.8	10	7.6	13	11.0
p = 1, t = 3, l = 4	10	5.5	12	7.1	13	8.9	18	13.9
	10	5.5	12	7.3	13	8.9	17	12.6
	10	5.5	10	6.5	15	10.2	16	12.0

Conclusion : the numbers increase slightly more than linearly with the number of the keywords.

Variation of the length of the keywords : l .

l =	2		3		4		5	
p = 1, t = 3, k = 3	7	5.7	9	6.3	12	7.2	16	8.9
	6	5.3	10	6.8	13	7.9	15	8.3
	6	5.2	9	6.2	11	6.3	16	8.9
	5	4.8	9	6.2	12	7.2	15	8.3
	6	5.3	10	6.8	12	7.3	15	8.3
p = 1, t = 3, k = 4	6	6.1	11	8.4	15	10.1	18	10.7
	7	6.5	9	7.3	16	11.1	19	11.9
	6	6.0	11	8.5	16	11.0	18	10.9
	7	6.7	11	8.4	13	8.7	17	10.4
	6	6.1	11	8.4	15	10.1	19	11.9

Conclusion : exponential behaviour should have become visible especially with regard to the length of the keywords; we can only conclude a behaviour between linear and quadratic.

III. type-4 grammars with negations and don't cares.

Above we have seen that the difference in complexity between type-4 grammars with and without pattern matching is small. In this subsection we will therefore omit the subformalisms for patterns and will concentrate on negations and don't cares. We will show how to write the classical NP-problem of satisfiability into a grammar which is written in the unifying formalism, using terminals, don't cares and negations.

We recapitulate the following definitions (Hopcroft and Ullman, 1979, p. 324 and 328) .

A Boolean formula is in conjunctive normal form (CNF) if it is the logical AND of clauses, which are the logical OR of literals. We say the formula is in k-CNF if each clause has exactly k literals. For example, $(x \vee y) \wedge (x' \vee z) \wedge (y \vee z')$ is in 2-CNF. An expression is satisfiable if there is some assignment of 0's and 1's to the variables that gives the expression the value 1. The satisfiability problem is to determine, given a Boolean expression, whether it is satisfiable.

It has been proven that L_{3sat}, the satisfiability problem for 3-CNF expressions, is NP-complete, just like the general satisfiability problem. The length of the problem is the length of the Boolean expression.

We take as an example a formula in 3-CNF :

$(a \vee c' \vee d) \wedge (a \vee b' \vee c) \wedge (a' \vee b \vee d)$ and rewrite it as

$((a' \wedge c \wedge d') \vee (a' \wedge b \wedge c') \vee (a \wedge b' \wedge d'))'$.

The encoding in the unifying formalism of the satisfiability of this expression is made as follows. We create for the rewritten Boolean expression a FSA which will read (conceptually) the four values for a, b, c and d, in that sequence. If the FSA will reach an accepting state than the expression is satisfied, otherwise it is not. But for the encoding of the satisfiability problem we do not have to read the 4 values : if an FSA can be constructed with one or more accepting states then the Boolean expression for which it is constructed is satisfiable (these states are always reachable, by construction).
In order to write the Boolean expression as a grammar in the unifying formalism we write each sub-expression as a string with alphabetical ordering of the symbols. Characters which are not present in a sub-expression are denoted by a don't care. The expression in our example becomes the grammar:

S :: '[a'*cd' | a'bc'* | ab'*d']1.

For this grammar we construct a set of itemsets which may be transformed into a FSA.
Our main conclusion can be that the complexity of GenLRsets for the subformalisms with don't cares and negations is in NP.
It is tempting to investigate the reason of the potential exponentiality of the rewritten problem. In the process of itemset generation a newly created set is compared with already existing sets for equality. This is the same technique as was used by Earley who brought down the recognition time for cfg's from exponential to polynomial (which technique is also used in the PTA). We will therefore perform a short analysis with the same parameters as were used in our former analysis in subsection II. Therefore, we are confronted with a grammar of the form

S :: '[E_1 | E_2 | .. | E_k]1.

Each E has the length l. In essence the size t of the alphabet is 3 : the position of a character in a keyword determines which character it is; it is present in its positive form or in its negated form or it is absent, in which case it is coded as a don't care. For our analysis we assume that there is an equal probability for these 3 possibilities.
The algorithm GenLRsets starts with an itemset which contains the items .'E_1 , .'E_2 .. .'E_k. The two possible shift symbols are a and a'. All items will travel over each symbol. For 1/3 of them the negdot will change in a don't care dot. The last kind of items can be furher ignored because their failure will only depend on other negdot items. Therefore 2/3 of the items will travel along a shift symbol. For the 2nd position 2 itemsets will be created with each 2/3.k items. For the 3rd position 2^2 itemsets will be created with each $(2/3)^2$.k items. Therefore in the j'th position 2^{j-1} itemsets with each $p_j = (2/3)^{j-1}.k$ items. However,
- before the l'th position there can be only 3 different items with a negdot which will behave different with respect to the last character; on the (l-1)'th position there are 3^2 possible items with a different future; on the j'th position therefore 3^{l-j+1} possible items
- after the j'th position there can only be $\binom{k}{p}$ different itemsets. The total number of work that has to be done for the j'th position depends solely on the total number of items on that position. This is therefore

$$p_j \,.\, \min\left(\binom{k}{p_j} , 2^{j-1} \right) , \text{ with } p_j = \min\left((2/3)^{j-1}.k , 3^{l-j+1} \right).$$

This may be summed for j = 1..l to get the total amount of work. The appearance of the minimum function accounts for the untransparancy of this expression and gives an explanation for the fact that some instances of this NP-problem do not show an exponential character.

Supscanner.

The module Supscanner creates for each itemset from the LR-table a jump-table. The work to be done is proportional to the number of symbols which may be expected for the itemset, which is a subset of the total number of symbols in the grammar. (The jump-table for terminals consists of ranges of characters. It could be replaced by, for instance, by an array of size 256 or by more refined sets of tables, like proposed in (Aho, Sethi, Ullman, 1986).)

SupLRcode.

The module SupLRcode transforms for each itemset the entries for a symbol in the LR-table into instructions and places a pointer to the set of instructions in the scanner. This operation takes time proportional to the number of reductions, for which the maximum is the number of rhs's in the grammar.
The generation of code for variables takes time proportional to the length of the expressions in which they appear.
The possibility of the sharing of code is investigated with the aid of hashcoding.

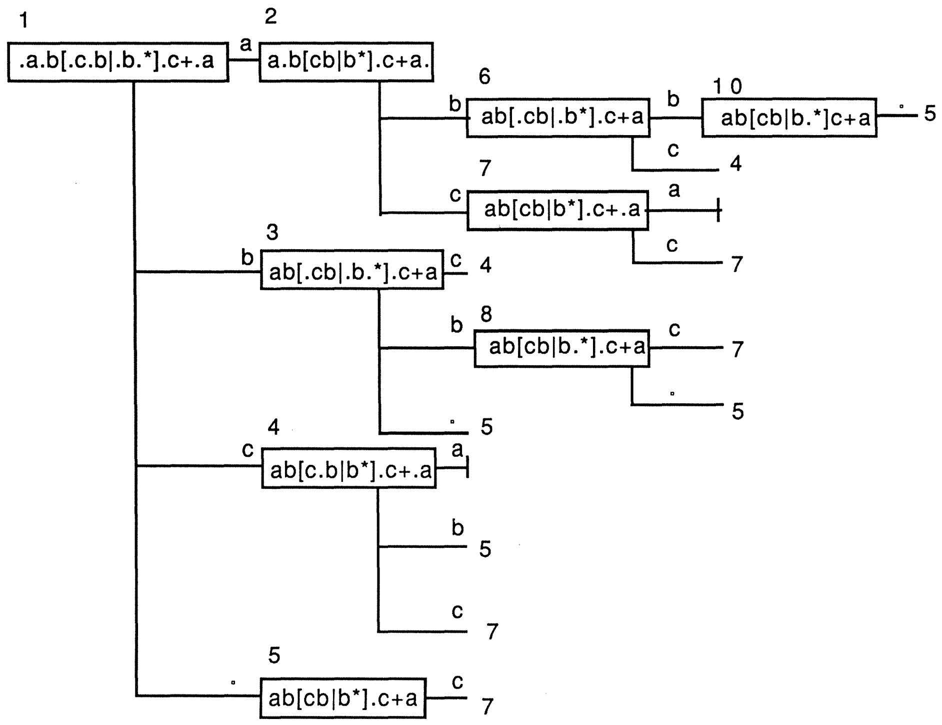
All modules together.

The complexity of the compiler as a whole is obviously determined by the complexity of the algorithm GenLRsets, which is exponential.
When itemsets are constructed only when the PTA needs the code for it ("lazy compilation") then the complexity of the algorithm GenLRsets becomes linear with the input to the runtime system plus the symbols which are reduced at a lhs.

7.2 Other applications of the algorithms within the compiler

7.2.1 Constructing position-trees

Up till now we treated the unifying formalism with the (enhanced) LR parser construction method. But it is also possible to exercise the same treatment on a text instead of on a grammar. In that way we may construct "Position trees" in order to locate all appearances of a pattern in a time proportional to the length of the pattern. Nodes in the tree will be identified by constructed itemsets.
The advantage is that all formalisms for a side of a rule are allowed for a text. The disadvantage is that the construction time may be, in worstcase, exponential.
We present the idea by an example.
Suppose the text is "ab[cb|b*][c]+a" (the "*" denotes a don't care). The starting itemset contains all possible items.



The length of the written text is 12 characters. Only 11 itemsets had to be constructed. 12 initially constructed itemsets did already exist.

7.2.2 Pattern matching of ill-formed input

In chapter 1 we indicated the problem of the treatment of ill-formed input. Depending on the parsing strategy a number of solutions have been proposed. However, these solutions suffer from an overhead in runtime. In order to start an investigation on the efficient treatment of ill-formed input in the system Parspat we tried a number of alternative approaches by an extension of the LR parser generation algorithm or of the PTA. One of the approaches concentrates on the problem of string matching with k mismatches, i.e. with at most k locations in which the pattern and the text have different symbols. Galil and Giancarlo (1986) published an algorithm (as an improvement of earlier algorithms which they reference) for the case of the matching of one keyword with length m in a text of length n with at most k mismatches. They achieve a time performance of $O(m.\log m+k.n)$ for general alphabets and of $O(m+k.n)$ for alphabets whose size is fixed. The algorithm uses $O(m)$ space.

We will now show how the runtime bound can be improved by making use of our technique for the handling of don't cares. We don't have to extend the algorithm GenLRsets or the PTA. The advantage of our approach is that the solution immediately works for all

subformalisms of the unifying formalism for which a FSA can be constructed (like in section 7.2.1) We conjecture that it can be extended to the whole unifying formalism.
The idea is that recognizers for ill-formed input may be constructed by systematically replacing terminals in a grammar by don't cares. As an example we demonstrate how to construct a pattern matcher for ill-formed input for a number of keywords.
For the case of one mismatch we add each keyword a number of times, each with a "don't care" in another position. If we are interested in 2 mismatches we add the keywords in versions where 2 characters are replaced by "don't cares", and so on. In general, for each allowed mismatch-construction we add the keyword a number of times with "don't cares" in appropriate positions. After compilation the runtime behaviour will still be linear.
We illustrate the principle of this method by the following example. Suppose we want to match the keywords "abc" and "bcd" with 0 or 1 mismatch. Then we construct a FSA for the grammar

S :: -, T.
T :: a, b, c | *, b, c | a, *, c | a, b, * | b, c, d | *, c, d | b, *, d | b, c, * .

Obviously, the generated FSA will have more states than the FSA for the grammar without don't cares. In order to get an impression of the multitude of generated states we perform an experiment analogous to the experiment in subsection 7.1.II. We add to the parameters already mentioned the parameter i with the following possible values

i = 0	no effect
i = 1	one mismatch allowed
i = 2	two mismatches allowed

and measure again the number of generated itemsets and the cpu-time for some values of the parameters t, k, l and p.

Variation of the allowed number of mismatches : i .

i =	0	1	2
p = 1, t = 3, k=3, l = 3	10	17	
	9	21	
	9	12	
p = 1, t = 3, k=3, l = 4	11	21	
	13	30	
	10	45	
p = 1, t = 3, k=3, l = 5	16	73	
	15	77	
	13	38	
p = 1, t = 9, k=2, l = 3	7	17	14
p = 1, t = 9, k=2, l = 4	10	39	100
p = 1, t = 9, k=2, l = 5	12	55	204
p = 1, t = 9, k=2, l = 6	14	53	
p = 1, t = 9, k=2, l = 7	16	89	

Conclusion : if we compare the number of itemsets for i=0 and i=1 we expect at most a multiplication with the length of the keywords l (because the original keyword is duplicated l times and because we saw earlier that the increase in keywords only influences the number of itemsets more or less linearly). For i = 2 we expect the factor l2. This is an upper bound because newly created itemsets may have a duplicate. The results conform to the expectation.

7.3 The complexity of on-line recognition for type-4 and type-3 grammars

In chapter 6 we showed how to generate a FSM for type-4 and -3 grammars, which obviously runs in linear time. In chapter 5 we explored the travelling rules for items in order to extend the algorithm for the creation of LR-tables for the following sub-formalisms :
- regular expressions
- concerning symbols :
    - ranges of terminal-symbols
    - tree-symbols
    - "don't cares"
    - "arb's" and "lines"
- concerning Boolean expressions :
    - coordination of grammar-rules and the Boolean "and"
    - negations.

For these extensions we obtained automatically a linear runtime behaviour. We compare this behaviour with the behaviour of algorithms from the literature, which were listed in section 2.6 and which were developed each for only some of our subformalisms:
- regular expressions : same runtime
- ranges of terminal-symbols : same runtime
- tree-symbols : sublinear runtime of our algorithms when in the input a pointer is maintained at an opening bracket to its corresponding closing bracket, else a linear runtime; the sublinear behaviour is an improvement over existing algorithms
- "don't cares" : linear runtime of our algorithms. The fastest algorithm for the recognition of one keyword with a "don't care" character was given by Fischer and Paterson (1974) with a time function of $O(m.(\log n)^2. \log \log n)$, with m as the length of the keyword and n the length of the input. Up till now no linear time algorithm was known. The recognition in linear time of keywords in a textstring, where both keywords and text may contain "don't cares" is, theoretically, related to "and-or multiplication" (Aho, Hopcroft, Ullman, 1974, p. 358).
- "arb's" and "lines" : "arb's" are not identified in the literature other than in the programming language Snobol. The linear treatment of the "line" is suggested by Aho and Corasick (1975) as a cascade of FSA's. We unified the treatment of the "line" at the start of a grammar-rule with the treatment of pattern matching. Only the algorithm of Boyer and Moore (1977) for the pattern matching of one keyword runs in sublinear time. We will discuss the relation of the algorithm of Aho and Corasick with our algorithms. After that we will discuss an extension of the algorithm GenLRsets in order to obtain the same sublinear runtime behaviour as the algorithm of Boyer and Moore, but then with an indefinite number of keywords, don't care symbols and ranges of symbols.
- cooperation of grammar-rules and the Boolean "and" : these operators are not treated in the literature in the context of pattern matching.
- negations : we are only aware of the algorithms of Ken-Chih Liu (1981) who treated Boolean negation of strings as an extension to the formalism of cfg's; in any case the runtime behaviour of his algorithms is more than linear.

7.3.1 Relation to the algorithm of Aho and Corasick

Two groups of authors which treated the problem of pattern matching, (Knuth, Morris and Pratt, 1977) and (Aho and Corasick, 1975), suggested that their method was akin to the construction of itemsets in the LR-parsing technique (originally developed by Knuth).

Aho and Corasick (1975) ("A&C" for short) described a method to transform a set of keywords into a FSA with an output function, operating in linear time. At the end of their paper they mention the problem of the treatment of "don't cares" in linear time. They also related the problem of pattern matching to LR parsing.
A&C make use of a "failure function" in order to compute points in the FSA where recognition can be resumed after the mismatch of all keywords up to a specific point. In our approach we avoid the use of a failure function. Our treatment of "don't care" characters is not limited to keywords with terminals (which we called type-4 grammars), but works together with all the subformalisms in the unifying formalism. Nevertheless, our research started with the elegant paper of A&C and we want to indicate how their algorithm could have been extended for the treatment of "don't care" characters in linear time. One more reason is the complexity of the creation of itemsets. The upper bound of A&C for space is lower than our upper bound (which is in principle exponential). A comparison of the two methods may, perhaps, shed more light on the complexity of itemset creation.

The algorithm of A&C consists of 3 parts :
1. the construction of a goto-function,
2. the construction of a failure-function,
3. the construction of a FSA.

In order to extend the algorithm for the treatment of don't cares only the construction of the goto-function has to be adapted.
The goto-function is realized by the construction of a trie for all the keywords. Nodes in the trie are identified as the states for the FSA that has to be constructed. In the algorithm of A&C keywords are added one after the other to the trie. If the trie should have been constructed by treating all keywords in parallel, like it is done in the LR-approach, then A&C could have proceeded in the following way.
In our terminology a keyword travels along a path in the trie. The paths are projected by the keywords themselves. If a keyword cannot longer share a path with another keyword it projects a continuation of the path at the current node and continues travelling along that path. If a keyword ceases it is added to the output function of the current state.
What happens when we want to spell out a keyword containing "don't cares" ? Suppose we are on its path in state s, where a number of continuation paths are possible, for a number of symbols. With the "don't care" character it is possible to follow each of these paths. We therefore put the current keyword on each available goto path and create an additional goto path for all the symbols for which no other keyword has a continuation. The trick is therefore to let keywords travel along more than one path. This is akin to the LR itemset construction method where we formulated this schema in the travelling rules for items.
With the LR method it is not necessary to construct failure-states, like in the A&C algorithm. In the case of pattern matching it produces directly a FSA. The algorithm of A&C constructs the same FSA with output as does our algorithm GenLRsets does. The algorithm of A&C

does this in linear time. It seems obvious that our algorithm performs in the same manner, which is supported by the measurements we did in the subsection on pattern matching.

7.3.2 Relation to the algorithm of Knuth, Morris and Pratt

Knuth, Morris and Pratt (1977) described another algorithm for matching single keywords in linear time (which was corrected by Rytter (1980)). They construct, like A&C did, a failure function, but without constructing a FSA. Their method is difficult to extend to, for instance, the matching of more keywords and the treatment of don't cares.

7.3.3 Relation to the algorithm of Boyer and Moore

Boyer and Moore (1977) ("B&M" for short) described how to preprocess one keyword in order to get, on the average, a less than linear runtime behaviour. This behaviour is possible by not inspecting every character in the input. The preprocessing takes time linear to the length of the keyword. The question arises whether their method can be generalized in order to
- improve our algorithms to get a sublinear runtime in the case of pattern matching,
- extend the improvement towards the treatment of (a) more than one keyword, (b) arb's, (c) negations of symbols and (d) subclasses of type-2, -1, -0 and transduction-grammars.

We will show how to answer that question in the affirmative for (a) and (b). It can simply be extended to incorporate case (c).We conjecture that the improvement can also be extended to (d).

The method of Boyer and Moore has been improved a number of times, especially for the worstcase behaviour of O(n2). The latest result which came to our attention was mentioned by Apostolico and Giancarlo (1986). They were able to derive an upper bound of 2n for the number of character comparisons in runtime while maintaining a linear preprocessing time. Our new upper bound is n comparisons in runtime, for more than one keywords written with don't cares. We discuss our method globally as an extension of the algorithm GenLRsets and then in detail by an algorithm which stands on its own.

Principle of the extension of the algorithm GenLRsets.

In order to obtain a better performance than Apostolico and Giancarlo did (and for all subformalisms which we mentioned above) we extend the algorithm GenLRsets as follows:
- with regard to the datastructures :
    - construct separate itemsets which indicate the effect of the skipping of characters in the input; such an itemset is called a "Skip_itemset"
    - add to each itemset a "mask" which describes, relative to the current position in the input, which characters have been skipped and which characters already have been read; this mask will become an integral part of the itemset to which it belongs and will be taken into account when two itemsets are tested for equality; in the algorithm the mask will have two functions:
        - it determines which items of the set of all possible items may be present in the itemset
        - it gives the necessary information to create Skip_itemset

- add to a FSA (with the usual reporting function) a skip instruction for the skipping of characters in the input

- with regard to the algorithm GenLRsets:
  - widen the concept of shift: if possible create from an itemset a Skip_itemset with the dot in the mask shifted to the right as far as possible (a "positive skip") or shifted to the left as short as possible (a "negative skip")
  - widen the concept of reduction : remove an item from a created itemset when it is fully matched by the mask; report it together with the calculated position in the input
  - widen the concept of the addition of closure items : add items to an itemset which are permitted by the mask, which are not already present and which are not fully matched by the mask (otherwise they should already have been reported)
- with regard to the algorithm SupLRcode:
  - in addition to the creation of code for a FSA generate skip instructions from the set of generated itemsets (which includes Skip_itemset's)

The algorithm BM.

Notations.
We will use the following notations :
- for a mask: u ∘ v , where u and v are strings of (possibly complemented) sets of terminals or of the reserved symbol ∅; substrings of u and v are sometimes written as $m_1..m_p$. Patterns within a mask are denoted with 0's and 1's, where a 1 stands for any non-∅ element of a mask
- for an item: $\alpha . \beta$; substrings of α and β are sometimes written as $a_1..a_q$
- the symbol '*' is reserved for the don't care symbol, the symbol '.' for the rest symbol (the rest symbol denotes in an itemset the set of terminals for which no shift is possible)
- a set with the elements a and b is denoted by [a, b]

Principles of operation.
A mask u ∘ v determines which items are permitted in the itemset I to which it belongs. This leads to the following principles according to which the algorithm is structured :
- shift the mask always as far as possible to the right, that is as far as is allowed for the shortest right part of the items; therefore

$$|v| = \min(|\beta|) \text{ for all } \beta \text{ of the items in I}$$

- shift the mask back as short as possible to reach the next unmatched character in the α part of the mask
- u need not be longer than the number of characters at the left side of the dot in any item (there is no remembrance of history when it is no longer necessary); therefore

$$|u| \leq \max(|\alpha|) \text{ for all } \alpha \text{ of the items in I}$$

The extensions to the algorithm GenLRsets follow from these principles. The algorithm GenLRsets makes use of optimizations which are prepared in the module Preparesets. Some of them have to be extended because of the introduction of the mask. On the other hand some parts of it are not needed because of the more simple formalism with which we are dealing. We therefore prefer to present the algorithm on its own.

Datastructures.

The types of datastructures are listed below. As usual we give no declarations for individual variables : their type is included in their name.

$\mathbb{B}$ denotes boolean, $\mathbb{N}$ denotes integer and $\mathbb{C}$ denotes a terminal symbol (in principle the number of terminals does not need to be finite).

A list of elements may also be indexed like an array.

```
Itemsets ::        (Itemset)*.
Itemset ::         (mask, items, items_reported, is_a_skipstate, [nr_of_skips_to_skipstate,
                   skip_state | entries ]1).
is_a_skipstate ::       B.
mask ::            ( msymbols, dotposition). { a dotposition counts from the left }
msymbols ::        ( mset )*.
mset ::            (complement, symbolset). { if complement = true then the Boolean
                     complement of symbolset is denoted }
symbolset ::       ( C)*.          { an empty symbolset corresponds with the symbolic ∅ }
items ::           ( item )*.
item ::            ( (C)+, dotposition).
nr_of_skips_to_skipstate | skipstate | dotposition ::   N.
entries ::         (C, actions)*, (rest_symbol, actions).
actions ::         (report, nr_of_skips, gotostate).
report | nr_of_skips | #created_itemsets | gotostate ::          N.
```

Algorithm BM

```
{ initialize }
- read the keywords
- prepare the set of all items
- construct the set starting_items of items with the dot at the leftmost position
- construct for each symbol a in the keywords the set items_before_symbol[a] of items with
  the dot before symbol a
- Itemsets[1] := ∅
- Itemsets[1].is_a_skipstate := false
- Itemsets[1].mask := (0, 0)
- Itemsets[1]items := starting items

- adjust_mask(Itemsets[1])
- put a reference to Itemsets[1] in the queue; #created_itemsets := 1

  { treat next itemset in the queue }
- while the queue is not empty do
  - take the reference to the next itemset from the queue and call the itemset :
    Current_itemset
  - if not Current_itemset.is_a_skipstate
   { create skip itemset }
     { suppose mask of Current_itemset =  u ∘ v }
    - if u = [ 0 | 1]* 0 1^n with n > 0
```

```
        - nr_of_skips := -n-1
    - else
        - if v = [ 0 | 1]$^m$ 0 with m $\geq$ 0
            - nr_of_skips := m
    - if nr_of_skips = 0
        - Current_itemset.is_a_skipstate := true  { nothing to skip; treat Current_itemset
                again, but then as a skipstate }
    - else
        { create skipitemset }
        - Skip_itemset := $\varnothing$
        - Skip_itemset.is_a_skipstate := true
        - shift_dot_in_mask (Current_itemset.mask, nr_of_skips, Skip_itemset.mask)
        - complete_itemset(Skip_itemset, skip_itemsetnr)
        - Current_itemset.nr_of_skips_to_skipstate := nr_of_skips
        - Current_itemset.skip_state := skip_itemsetnr
        - remove the reference to Current_itemset from the queue

- else { Current_itemset.is_a_skipstate }
    { **create** from Current_itemset **all possible New_itemset's** with their mask }
    - collect  in the set shift_symbols all symbols, excluding the don'tcare, on which
      a shift is possible
    - shift_symbols := shift_symbols  $\cup$  rest_symbol   { the reserved "rest_symbol"
      denotes the complement of the set of all symbols for which in Current_itemset a shift
      can be generated; it is alternatively written as "." }
    - for each symbol a $\in$ shift_symbols do
        - New_itemset := $\varnothing$
        - New_itemset.is_a_skipstate := false
        { shift mask over symbol }
        - New_itemset.mask := Current_itemset.mask
        - if a = rest_symbol
            - New_itemset.mask.msymbols[dotposition] := ({complement=}true,
                                                        shift_symbols)
        - else
            - New_itemset.mask.msymbols[dotposition] := ({complement=}false, [a])
        - New_itemset.mask.dotposition := New_itemset.mask.dotposition + 1
        - Temp := (Current_itemset $\cap$ ( items_before_symbol[a]  $\cup$
                        items_before_symbol[*] )
        - for each item i in Temp do
            - if match_of_item_with_mask(successor item of i, New_itemset.mask,
                        {complete=}true)
                - Current_itemset.entries[a, report] := number of keyword of item i
                - Current_itemset.items_reported := Current_itemset.items_reported  $\cup$ [i]
            - else
                - New_itemset.items := New_itemset.items  $\cup$  [successor item of i]
        - adjust_mask(New_itemset)
        - complete_itemset(New_itemset, itemsetnr)
        - remove the reference to Current_itemset from the queue
- for all itemsets replace all shifts to itemsets which are not a skipstate and for which a skip
  exists to a skipstate s, by a shift to skipstate s, with an account of the number of skips
{ end of  BM }

procedure adjust_mask(Itemset)
{ suppose the current mask of Itemset is $u_1.v_1$ and the new mask is $u_2.v_2$ }
- determine for Itemset the sizes $lu_2$ of $u_2$ and $lv_2$ of $v_2$ such that
        $lu_2 := \min( |u_1|, \max( |\alpha| ) )$ for all $\alpha$ of the items $\alpha.\beta$ in Itemset
        $lv_2 := \max( |v_1|, \min( |\beta| ) )$ for all $\beta$ of the items $\alpha.\beta$ in Itemset
- $u_2$ := the suffix of $u_1$ with length $lu_2$
- $v_2$ := the prefix of $v_1$ with length $lv_2$
- fill vacant positions in $v_2$ with a $\varnothing$
{ end of adjust_mask }

Boolean function match_char(item_symbol, mask_mset, complete)
- match_char := ( ( item_symbol $\in$ mask_mset.symbolset ) and
                        (not mask_mset.complement) )
        or      ( ( item_symbol $\notin$ mask_mset.symbolset ) and (mask_mset.complement) )
        or      ( ( mask_mset.symbolset = $\varnothing$) and not complete )
{ end of match_char }

Boolean function match_of_item_with_mask(item, mask, complete)
{ suppose item = $\alpha.\beta$ and mask = u.v
            u       v
        ------- . -----
            α       β
            ---- . ----------  }
- return true  if
        - $| \alpha | \leq | u |$
  and   - ( $| v | \leq | \beta |$ and not complete) or ( $| v | = | \beta |$ and complete)
  and   for each position p in $\alpha^T$ match_char( $\alpha^T[p]$, $u^T[p]$, complete ) ($^T$ means : reversed)
  and   for each position p in v match_char( $\beta[p]$, v[p], complete )
{ end of match_of_item_with_mask }

Procedure fill_skipitemset_according_to_mask(Skip_itemset)
- for all non-reducing items i do
   - if match_of_item_with_mask(i, Skip_itemset.mask, false)
      - if (i $\notin$ Skip_itemset.items) and (i $\notin$ Skip_itemset.items_reported)
         -  Skip_itemset.items := Skip_itemset.items $\cup$ i
{ end of fill_skipitemset_according_to_mask }

Procedure adjust_mask_and_add_to_queue(itemset, itemsetnr)
- if itemset.items = $\varnothing$
   - itemsetnr := 1
- else
   - adjust_mask(itemset)
   - if itemset not already constructed
```

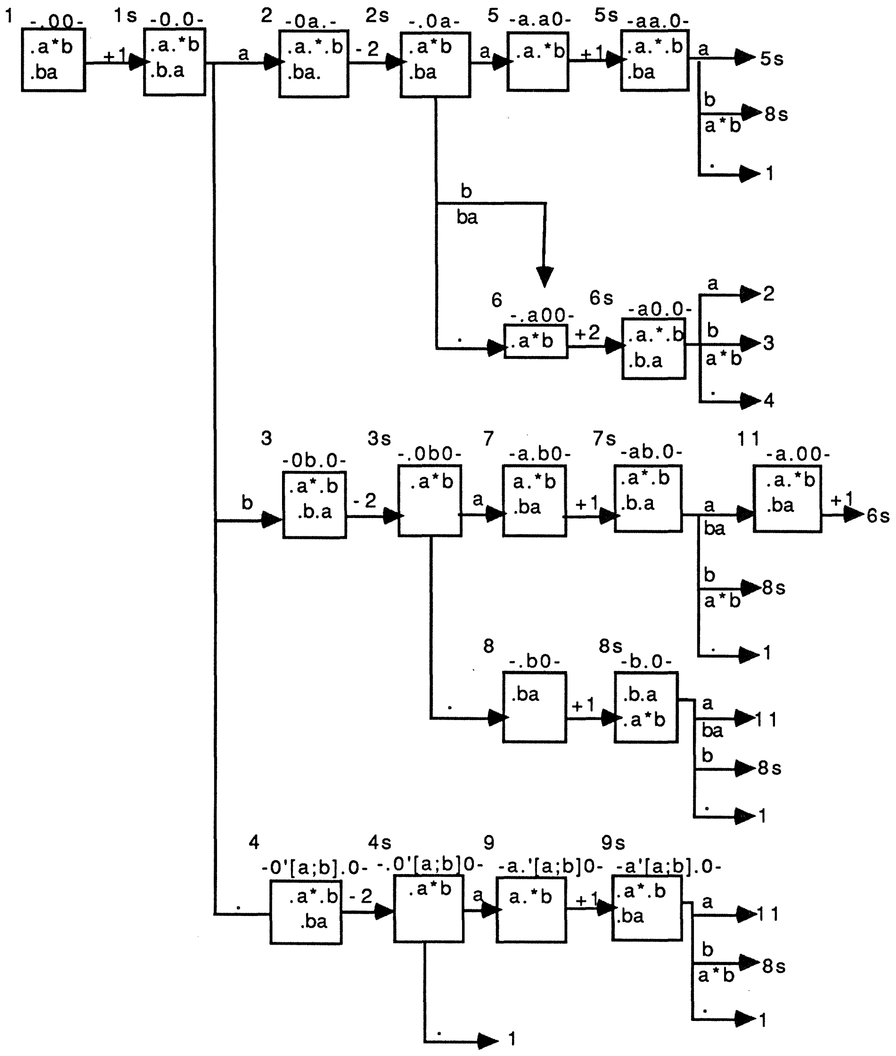
```
        - add reference to itemset to queue
        - itemsetnr := #created_itemsets := #created_itemsets+1
        - Itemsets[#created_itemsets] := itemset
    - else itemsetnr := number of the matched itemset
{ end of adjust_mask_and_add_to_queue }

Procedure complete_itemset(itemset, itemsetnr)
- fill_skipitemset_according_to_mask(itemset)
- adjust_mask_and_add_to_queue(itemset, itemsetnr)
{ end of complete_itemset }

Procedure shift_dot_in_mask(mask_old, s, mask_new)
- if s > 0
    { suppose mask_old = $u \circ m_1..m_{s+1}\ \emptyset$ }
    - mask_new := $u\ m_1...m_{s+1} \circ \emptyset$
- else { s < 0 }
    { mask_old has to be of the form $m_1...m_p\ \emptyset\ 1^s \circ v$ , with p >= 0, see below }
    - mask_new := $m_1...m_p \circ \emptyset\ 1^s\ v$
{ end shift_dot_in_mask }
```

Example.

We create an FSA with output and skip function for the keywords a*b and ba according to the algorithm as follows. (For better readability the masks are surrounded by a '-'.)



Generation of a FSA with output and skip function, on the average operating in less than linear time and in worst case inspecting each input character only once for the keywords: "a*b" and "ba" .

In the next table format we create immediately a goto to a skip-state and perform the corresponding skip directly (the reading of a character always implies the shift of the input head):

state	symbol	report	skip	goto state
1			+1	1s
1s	a		-2	2s
	b		-2	3s
	.		-2	4s
2s	a		+1	5s
	b	ba	+2	6s
	.		+2	6s
3s	a		+1	7s
	.		+1	8s
4s	a		+1	9s
	.		+2	1s
5s	a			5s
	b	a*b		8s
	.		+1	1s
6s	a		-2	2s
	b	a*b	-2	3s
	.		-2	4s
7s	a	ba	+1	6s
	b	a*b		8s
	.		+1	1s
8s	a	ba	+1	6s
	b			8s
	.		+1	1s
9s	a		+1	6s
	b	a*b		8s
	.		+1	1s

Runtime complexity.

If n is the length of the input and m is the length of the shortest keyword, the upper bound for the number of inspected input characters will be n and the average will be n/m. No calculation at runtime is needed and each input character is inspected at most once.

Complexity of constructing itemsets.

We made use of a modification of the algorithm GenLRsets which runs, as we have seen, in time more or less linear in the size of the grammar. The number of possible itemsets will increase because a unique mask is added to an itemset. However, the mask governs the creation of new itemsets and the algorithm BM puts some constraints on the form which the mask can take. We will do some analysis on that point. After that we will present some average-cost measurements on the number of created itemsets.

Structure of the mask.

The following observations can be made on the structure of the mask. We will denote with the reserved character "1" the elements in the mask which are not ∅ and by "0" the elements which are ∅.

	negative skip	positive skip	no skip
Current_itemset	$[\,0 \mid 1]^*\ 0\ 1+ \circ v$	$1^* \circ [\,0 \mid 1]+\ 0$	$1^* \circ 0$
Skip_itemset	$[\,0 \mid 1]^* \circ 0\ 1+\ v$	$1^*\ [\,0 \mid 1]+ \circ 0$	n.a.
New_itemset	$[\,0 \mid 1]^*\ 1 \circ 1+\ v\ 0^*$	$1^*[\,0 \mid 1]+\ 1 \circ 0^*$	$1^*\ 1 \circ 0^*$
simplified :	$[\,0 \mid 1]^*\ 1 \circ 1+\ v\ 0^*$	$[\,0 \mid 1]+\ 1 \circ 0^*$	$1+ \circ 0^*$

Conclusions :
- v will be right-padded with 0's. Because v is initially 0^+ it will retain as a suffix the form 0^+.
- u will always end on a 1 (when not empty).
- therefore, the general form will be
    - for u : $u = [\,0 \mid 1]^*\ 1$ (when not empty)
    - for v : there is a recurrence relation : $v_1 = 0^+$, $v_{n+1} = 1+\ v_n\ 0^+$ or 0^*; this is satisfied by $v = 1^+\ 0^+$ or 0^*; therefore a more refined general form of v is $1^*\ 0^*$

If the refined general form of v is used in the schema above it becomes

	negative skip	positive skip	no skip
Current_itemset	$[\,0 \mid 1]^*\ 0\ 1^+ \circ 1^*\ 0^*$	$1^* \circ 1^*\ 0^*\ 0$	$1^* \circ 0$
Skip_itemset	$[\,0 \mid 1]^* \circ 0\ 1+\ 1^*\ 0^*$	$1^*\ 0^* \circ 0$	$1^* \circ 0$
New_itemset	$[\,0 \mid 1]^*\ 1 \circ 1+\ 0^*$	$1^*\ 0^*\ 1 \circ 0^*$	$1^+ \circ 0^*$

With this schema we may write a recurrence relation for u :

		negative skip	positive skip	no skip
u_1	=	ε		
u_n	=	$[\,0 \mid 1]^*\ 0\ 1^+$	1^*	1^*
u_{n+1}	=	$[\,0 \mid 1]^*\ 1$	$1^*0^*\ 1$	1^+

u may therefore be refined to the unifying pattern $1^*0^*1^*$. Substitution of this pattern in the schema does not lead to a further refinement.
The general form of the mask $u \circ v$ for itemsets is therefore $1^*0^*1^* \circ 1^*\ 0^*$. Skip-itemsets will always have a 0 after the dot.

Some measurements.

We programmed the algorithm BM and did some measurements according to the lines of section 7.1. The same program was used for the generation of grammars. We varied the parameters t (size of the alphabet), k (number of keywords) and l (length of keywords) and measured the number of generated itemsets, once for each combination of parameters. The results are as follows.

Variation of the size of the alphabet : t

k=1, l=	3	4	5	6	7	8
t=2	6	15	23	39	44	54
t=9	7	13	21	32	42	78

Conclusion: there is nearly no influence of the size of the alphabet.

Variation of the number of keywords : k

t=9, l=	3	4	5	6	7	8
k=1	7	13	21	32	42	78
k=2	13	25	44	69	88	114
k=3	19	42	64	95	141	194
k=4	28	55	87	130	202	255

Conclusion: there seems to be a linear correspondence of k with the number of generated itemsets.

Variation of the length of the keywords : l. The same table can be used as before.
Conclusion: there is a more than linear correspondence. It seems not to be exponential, but there is a stronger influence of the length of the keywords than is the case with the unmodified algorithm GenLRsets.

7.4 The complexity of on-line recognition and parsing of type-2 grammars

7.4.1 Within the system Parspat

When cfg's are parsed with the PTA only the L-dag is used. In chapter 6 we showed that the compiler itself may be used as an Earley parser, by setting the switch "interactive" to true and the switch "multi" to false. Both parsers run, in general, in time $O(n^3)$, but the first parser runs much faster then the second because all possible itemsets are already constructed by the compiler. In between the two is the behaviour when the option "lazy compilation" is used : itemsets are only constructed by the compiler when the code for the corresponding states are needed by the PTA.

The space complexity of the PTA is $O(n^2)$, the same as for Earley's method, because in a dagnode we store with each item the references to its return nodes (datastructure "poplist"). With each such reference a number of alternative paths (the datastructure "infolist") in the parse forest are associated. But this number can not be more than the number of admissable alternatives in the rhs of the corresponding grammar rule, which is a constant, even when the rule contains alternatives within regular expressions. This is the reason of the efficiency of the online representation of the parse forest. For each item in a dagnode, and for each returnstate which belongs to that item, the description for a path in the parse-forest is maintained. On a LSTK instruction only a finite amount of work is done for each of these infolists. On a LRDCF, PLRDCF or LRDCS instruction nothing extra has to be done to maintain the parse-forest : the infolist is simply attached to the created connector. For cfg's the information in a path concerns reports, builds, variables and the (alternative) parse(s) for the nonterminal at the lhs.

Towards linear time recognition of cfg's.

We have already seen in chapter 6 that for the sub-class of cfg's which generate a regular language the compiler generates a FSA. This was done by making use of the concept of "compatibility calculation". The question arises whether this concept can also be used in the PTA in order to obtain an improvement in runtime for general cfg's. This question is of some importance, as witnessed by the following quotation from Knuth, Morris and Pratt (1977) : "One of the most outstanding unsolved questions in the theory of computational complexity is the problem of how long it takes to determine whether or not a given string of length n belongs to a given context-free language. For many years the best upper bound for this problem was $O(n^3)$ in a general context-free language as $n \rightarrow \infty$; L.G. Valiant has recently lowered this to $O(n^{\ln 7})$.[1] On the other hand, the problem isn't known to require more than order n units of time for any particular language. This big gap (..) deserves to be closed, and hardly anyone believes that the final answer will be $O(n)$."
It seems that the authors were too optimistic with their statement about the linear time recognition of any particular language. Probably they were not aware of the "hardest context-free language" of Greibach (1973) which is a language encoding of any structural description of any well formed input sentence according to any arbitrary cf grammar. If the recognition time of this language is proven to be linear then all cf grammars may be recognised in linear time.
However, up till now no such proof exists. It is desirable that such a proof will be given in a constructive way by the presentation of an algorithm for the recognition (and eventually parsing) of general context-free grammars.

The question raised by Knuth et al. can be placed in the more general context of chapter 1. Ideally the formalism in which a problem is expressed should not contribute to the complexity of solving the problem. If we know that a problem can be solved with less complexity when it is stated in another formalism then we get the opportunity to compare in more detail the two corresponding processes. Our goal is to improve in this way the algorithms within the system Parspat.

In order to explore how an answer can be given to the question of the possibility of linear time recognition of cfg's we start our discussion with a recapitulation of some results about the complexity of cf recognition and parsing. We will analyse some worstcase languages and grammars from the literature and will try to discover the possible reasons for the worstcase behaviour. For each of the worstcase grammars we will show how they behave when they are treated by the Earley algorithm when it is enriched by the calculation of compatibility. This will be followed by the development of a cf grammar for the "most worstcase" language of Greibach and by a proposal for the extension of the concept of compatibility calculation in order to obtain a linear runtime behaviour for this language.

7.4.2 Recapitulation of relevant results about cf recognition and parsing

The following results are relevant to our discussion :

[1] The present lower bound is $O(n2.38)$

Kasami (1965) : - The upper bound for the recognition time of a linear cfl is O(n2) .

Earley (1970) : - The upper bound for the online recognition time of an arbitrary context-free grammar is $O(n^3)$.

Ruzzo (1979) : - Parsing strings of length n is harder than recognizing such strings by a factor of only O(log n), at most. The same is true for linear and/or unambiguous cf-languages

Ruzzo (1979) : - The time to multiply $n^{1/2}$ x $n^{1/2}$ Boolean matrices is a lower bound on the time to recognize all prefixes of a string (or do on-line recognition), which in turn is a lower bound on the time to generate a particular convenient representation of all parses of a string (in an unambiguous grammar). Thus these problems are solvable in linear time only if n x n Boolean matrix multiplication can be done in $O(n^2)$ (Ruzzo presents also a language encoding of boolean multiplication).

7.4.3 Extension of Earley's algorithm with compatibility calculation

In chapter 6 we introduced a modification of the algorithm GenLRsets. When an item reduces and the switch "multi=false" is used we did not create a reduce-entry in the LR-table but instead we copied the father item(s) in the father state(s) of the reducing item, with a shift on the reducing nonterminal. This is the technique of Earley, but applied to the algorithm GenLRsets. We extended the technique with the compatibility calculation between two itemsets which are referenced as father itemsets of an arbitrary item. We were able, with that extension, to generate an FSA for grammars which generate a regular language. The extension can be used also in the algorithm of Earley. We get its implementation for free when we use in the compiler of Parspat the switch "interactive=true" : in that case the algorithm GenLRsets constructs only a new itemset from the current one upon the reading of an input symbol.
When we use the compatibility checking in that case (switch "test_comp_sets=true") and still perform the check if the core of a created itemset already exists then the thus extended algorithm of Earley creates only a fixed number of itemsets when the grammar describes a regular language. When the grammar describes a cfl then the creation of new itemsets on input symbols continues, but we may expect that fewer references to father itemsets will be created.
Compatibility calculation takes, in general, O(n2) time. We will have a chance to obtain a O(n) time when we succeed in extending the algorithm for compatibility calculation in such a way that only one father has to be referenced for each item in an itemset.
We will first investigate the behaviour of some worstcase grammars.

7.4.4 Worstcase languages and grammars

In this subsection we will recall a number of worstcase context-free languages and grammars which have been mentioned in the literature, with their bounds on runtime recognition. We will classify them according to a number of criteria how they show their worstcase behaviour:

- the difference between recognition and parsing,
- the type of the language which is described,
- the kind of ambiguity of the grammar,
- the choice of the recognition strategy .

7.4.4.1 The difference between recognition and parsing

The difference between recognition and parsing is that in the latter case parse trees have to be represented. In section 7.4.2 we saw that the difference in runtime may be at most O(log n). General parsing strategies usually take into account the construction of parse trees. In the algorithm of Earley parsetrees are reconstructed after recognition has finished. In Tomita's and our algorithms the parse forest is constructed online. In our case we saw that this does not contribute to the complexity of recognition. We therefore disregard this factor in our search for an improvement of the time of recognition.

7.4.4.2 The type of the language which is described

Regular languages may be recognized in linear time. We already discussed our improvements with the aid of compatibility calculations. Below the effect is shown on some worstcase grammars.

Source : (Griffiths and Petrick, 1965), referenced by (Earley, 1970) and (Bouckaert c.s., 1975)
Purpose : worstcase grammars for cf recognition
Grammar1 : called GRE

X : a | X b | Y a
Y : e | Y d Y
with e.g. as input : $(ed)^n eabb$

Parspat: 1. a FSA is created with 6 states
2. when the compiler is used as an extended Earley parser also only 6 itemsets are created

Grammar2 : called UBDA

A : x | A A

Comment: Earley believes that UBDA is one of the few grammars which will take $O(n^3)$ runtime with his parsing method. In (Aho and Ullman, 1972, p. 165) an exercise is given to show that the number of distinct leftmost derivations of a^{n+1} according to UBDA is given by $^1/(n+1)\ (^{2n}n)$. But in section 6.4.1 we showed that with the aid of compatibility calculation a FSA for this grammar is constructed.

Gallaire tried to diminish the gap between lower and upper bounds for recognition by presenting an example of the coding of a problem into a linear cfl. For this problem a lower bound is known.

Source : (Gallaire, 1969)
Purpose : to show a lower bound for the recognition of cfl's with an on-line Turing machine with a language encoding of the pattern matching problem

Language : LGallaire = $\{\Sigma^*z\Sigma^*csu_1s...su_ts \mid t >=1, z \varepsilon \Sigma^*, |z| > 0, u_j \varepsilon \Sigma^*$ for $1 <= j <= t$ and $u_i = z^T$ for some i, $1 <= i <= t\}$(one or more of the keywords $u_1..u_t$, which are separated by a 's' from each other, are to match in reverse with some substring z in the text string $\Sigma^*z\Sigma^*$, which is separated by a 'c' from the keywords). This set is a linear cfl. Therefore the upper-bound = $O(n^2)$. Lower bound = $n^2/\log n$ on TM.
Comment : Gallaire proved his lower bound for a TM, but we know now that the problem of pattern matching of a number of keywords can be solved in linear time by a FSA. The gap therefore increases.
Grammar1 : Ggallaire1 (written as a linear cfg) :

```
S : 0 S | 1 S | A s | B s
A : A 0 | A 1 | A s | B s
B : 0 C 0 | 1 C 1
C : 0 C 0 | 1 C 1 | D
D : 0 D | 1 D | c E
E : s | s F
F : 0 F | 1 F | E
```

Grammar2 : Ggallaire1 rewritten as a ecfg : Ggallaire2:

```
S : A B s C
A : [a | b]*
B : a B a | b B b | a D a | b D b
D : A c C
C : [A s] *
```

Parspat: for both grammars the number of connectors is O(n), the number of infolists which had to be treated O(n2) and the number of elementary comparisons of returnstates in infolists O(n3). However, the cpu-time is about O(n2). This can only be explained by the fact that the addition of an element to a list with a check on its presence is a really basic operation which hardly contributes to the total processing time.

7.4.4.3 Inherent and temporal ambiguity

Novice grammar writers usually produce grammars with a large number of ambiguities. We call these ambiguities which are not intended "artificial ambiguities". For grammar writers who are only interested in recognition (with reporting) and not in parsing, the runtime behaviour of the parser should allow for the artificial ambiguities.
Takaoka and Amamiya (1975) developed an "ambiguity function" in order to calculate a measure for the potential ambiguous parses of a grammar. The following grammars are typical examples of the coding of a very simple problem in a grammar formalism. We present also the time of recognition as we measured it with the system Parspat.

Source : (Takaoka and Amamiya, 1975).
Purpose : to develop "ambiguity functions" to measure inherent ambiguity.
Knuth (1965) defines the
Language : LKnuth = $\{x \varepsilon \{a, b\}+ \mid$ equal numbers of a and b occur in x$\}$
Hopcroft and Ullman (1969) specify for L the following grammar
Grammar1 : GKnuth1:

```
S : b A | a B
A : a S | b A A | a
```

B : b S | a B B | b

Result: Takaoka and Amamiya calculate for this grammar an exponential ambiguity function.

Parspat : see Ggallaire.

Grammar2 : Takaoka and Amamiya specify for LKnuth also the grammar GKnuth2:

S : a B S | b A S | a B | b A
A : b A A | a
B : a B B | b

Result: This grammar is, according to their calculation, unambiguous.

Parspat : O(n)

Grammars can be written intentionally ambiguous. We then call them "inherently ambiguous". The following one is created very intentionally.

Source : (Hopcroft and Ullman, 1979).

Purpose : to exhibit a cfl for which every cfg is ambiguous.

Language: Lhopcroft = $a^n b^n c^m d^m + a^n b^m c^m d^n$, n>=1, m>=1.

Grammar: a straightforward grammar for Lhopcroft is Ghopcroft:

S : A, B | T
A : a, A, b | a, b
B : c, B, d | c, d
T : a, U, d
U : T | V
V : b, V, c | b, c

Parspat : see Ggallaire.

A grammar can be temporarily ambiguous. This may depend upon the parsing strategy. For instance, when a variable number of lookahead symbols is needed before a decision can be made about the reduction of a grammar rule, then temporary ambiguity manifests itself.

Source : (Graham, Harrison and Ruzzo, 1980)

Purpose: to show an unambiguous grammar with recognition time O(n3)

Grammar : GGraham :

S : A | B C
A : ε | a A B
B : b | c | a B
C : ε | b C D
D : c | b D

Language : $a^n (a^+(b+c))^n + (a^+(b+c)) b^m(b^+c)^m$, n, m >= 1

Parspat : see Ggallaire.

7.4.4.4 The choice of the recognition/parsing strategy

LR(k) and LL(k) parser generation concentrates on deterministic parsing by avoiding shift/reduce-reduce conflicts, using k symbols lookahead. The generated parsers operate in linear time. But some LR grammars will parse with Earley's method in O(n2) time. An even more striking difference is found in the recognition of palindromes. Palindromes cannot be described by a LR(k) grammar. Nevertheless, their exist efficient recognition strategies. Manacher (1975) describes an online algorithm for recognition of palindromes in linear time. The real time version is given by Galil (1978).

Language : Lpalind : $x\, x^T$, $x \in \{a, b\}+$
Grammar : Gpalind :
    S : a S a | b S b | a a | b b.
Parspat : see Ggallaire.

A linear time recognition algorithm is also given for the following grammar.

Source : (Knuth, Morris and Pratt, 1977)
Purpose : to show that palstars can be recognised in linear time by the KMP-algorithm
Language : Lpalstar = {x* | x ε Lpalind}
Grammar : Gpalstar :
    S : S S | a S a | b S b | a a | b b
Parspat : see Ggallaire.

7.4.4.5 The hardest context-free grammar : Ggreibach

Greibach (1973) presented a language (which we call Lgreibach) which encodes the problem of parsing cf grammars itself. In order to have a "most worstcase" cf grammar at hand and to study its structure we will develop the grammar Ggreibach which describes Lgreibach.
We recapitulate the definition of Lgreibach:
∇ Let $T = \{a_1, a_2, \underline{a}_1, \underline{a}_2, c, ©\}$ where $a_1, a_2, \underline{a}_1, \underline{a}_2$, c and © are all distinct. Let D be the context-free language generated by
    S : S S
    S : a_1 S $\underline{a}_1$
    S : a_2 S $\underline{a}_2$
    S : ε.
We call D a Dyck set on two letters. Let d be new and let
$L_0 = \{\varepsilon\} \cup \{x_1cy_1cz_1d...dx_ncy_ncz_nd \mid n\geq1, y_1...y_n \in$ ©D, $x_i, z_i \in T^*$
    for all i, $y_i \in \{a_1, a_2, \underline{a}_1, \underline{a}_2\}^*$ for $i\geq2\}$.
Thus the language L_0 selects one subword from each group set off by d's in such a way that the concatenation of the choices belongs to ©D, the Dyck set on two letters, preceded by ©. L_0 will encode derivations in a cf grammar. ∇
Greibach proves the following theorem :
∇If L is a cf language, there is a homomorphism h such that $L - \{\varepsilon\} = h^{-1}(L_0 - \{\varepsilon\})$.∇
Greibach assumes that L does not contain the empty word and that the grammar which describes L is written in the standard form, that is, the rules are of the form Z : a y, where a is a terminal and y contains neither terminals nor S.
h is formed as follows :
1. order the nonterminals in some way : $Y_1, ..., Y_n$, such that Y_1 = S.
2. define functions ξ, $\underline{\xi}$ from productions into $\{a_1, a_2, \underline{a}_1, \underline{a}_2\}^*$ as follows:
    - if p is the production Y_i : a, then $\underline{\xi}(p) = \underline{a}_1\underline{a}_2{}^i\underline{a}_1$
    - if p is the production Y_i : a $Y_{j1}...Y_{jm}$, $m \geq 1$, then

$\underline{\xi}(p) = \underline{a}_1\underline{a}_2{}^i\underline{a}_1 a_1 a_2{}^{j_m} a_1 \ldots a_1 a_2{}^{j_1} a_1$.

- if $i \neq 1$ then $\xi(p) = \underline{\xi}(p)$, if $i = 1$ then $\xi(p) = ©\ a_1a_2a_1\underline{\xi}(p)$

If $P_a = \{p_1, \ldots, p_m\}$ is the set of all productions whose rhs's start with a, then $h(a) = c\xi(p_1)c \ldots c\xi(p_m)cd$; without loss of generality one may assume $P_a \neq \emptyset$ for all a in Σ.

Greibach then shows that $h(w) = x_1cy_1cz_1d \ldots dx_ncy_ncz_nd$ with $k = |w|$, $y_i \in \{a_1, a_2, \underline{a}_1, \underline{a}_2\}^*$, $x_iz_i \in T^*$, $1 <= i <= k$ and $y_1 \ldots y_k \in ©D$ if and only if $y_1 \ldots y_k$ encodes the productions used in a left-to-right derivation of w from S.

We will now develop a grammar G_0 for L_0. Greibach notes that it is possible to construct G_0 such that the number of derivations of h(w) in G_0 is precisely the number of derivations of w in a grammar G for L; in this sense the homomorphism h "preserves multiplicities". Furthermore, G_0 can be so selected that an efficient parser for L_0 can be converted automatically into an efficient parser for L.
We are interested in the improvement of the upper bound for general cf recognition and will therefore give no attention to this "preservation of multiplicity". In that respect it should be more interesting when e.g. a linear cf grammar G_0 could be found. The recognition time would be in that case $O(n^2)$.

$L_0 = x_1\ c\ y_1\ c\ z_1\ d\ x_2\ c\ y_2\ c\ z_2\ d \ldots d\ x_n\ c\ y_nc\ z_n\ d$ describe 'x_1 c', 'x_2 c' etc. by $A = [\Sigma + c]^*$
'c z_1', 'c z_2' etc. by $B = [c\ \Sigma +]^*$

then $L_0 = A\ y_1\ B\ d\ A\ y_2\ B\ d\ A\ y_3\ B\ d \ldots d\ A\ y_n\ B\ d$ describe 'B d A' by C

then $L_0 = A\ y_1\ C\ y_2\ C\ y_3 \ldots C\ y_n\ B\ d$ describe $y_1Cy_2Cy_3 \ldots Cy_n$ by H

then S : A H B d => L_0 remains H

$y_1\ y_2\ y_3 \ldots y_n \in ©\ D$, where D, the Dyck set on two letters, can be described by the grammar:

Z : Z Z
Z : a_1 Z $\underline{a}_1$
Z : a_2 Z $\underline{a}_2$
Z : ε.

In a non-empty form this becomes :

Z : Z Z
Z : a_1 Z $\underline{a}_1$ | a_1 $\underline{a}_1$
Z : a_2 Z $\underline{a}_2$ | a_2 $\underline{a}_2$

We have to describe $H = y_1\ C\ y_2\ C\ y_3 \ldots C\ y_n$.
As a straightforward modification of the grammar for the Dyck set we may observe that 'C's are allowed between each terminal within the y's. We therefore get :

H : © Z
Z : Z [C] Z
Z : a_1 [C] Z [C] $\underline{a}_1$ | a_1 [C] $\underline{a}_1$
Z : a_2 [C] Z [C] $\underline{a}_2$ | a_2 [C] $\underline{a}_2$

Shorter :

H : © Z
L : Z [C]
Z : L Z
K : [C] | [C] L
Z : a_1 K $\underline{a}_1$ | a_2 K $\underline{a}_2$

Without regular expressions :

H : © Z
L : Z C | Z
Z : L Z
K : C | L | C L
Z : a_1 K $\underline{a}_1$ | a_2 K $\underline{a}_2$ | a_1 $\underline{a}_1$ | a_2 $\underline{a}_2$

These are the ingredients for G_0. The nonterminals A and B can be written out in a straightforward way. When we choose

for a_1	the character [
for $\underline{a}_1$	the character]
for a_2	the character (
for $\underline{a}_2$	the character)

we get :

Grammar : Ggreibach :

S : A H B d
C : B d A
A : A E c | ε
B : B c E | ε
E : I | E I
I : { | } | (|)
H : © Z
L : Z C | Z
Z : L Z
K : C | L | C L
Z : (K) | { K } | ( ) | { }

Parspat : see Ggallaire.

7.4.5 Evaluation of the results

The grammar Ggreibach does not look much different from other worstcase grammars which operate in cubic time. Central in the development of the grammar was the grammar description of the Dyck set on two letters, which is the same as for Palstars.
After inspection of the former grammars we have come to the conclusion that grammars which describe a regular language will run in linear time when the technique for compatibility calculation is used, but that grammars with infix recursion run in cubic time. In order to tackle the problem of the decrease of runtime we do not have to use Ggreibach : even the simpler grammars for the recognition of palindromes and palstars exhibit the cubic behaviour. Moreover, for the languages which they generate other strategies exist for the recognition in linear or in real time. It is therefore advantageous to concentrate on the

improvement of Earley's algorithm, to which we already added the compatibility calculation for the case of palindromes or Palstars.

7.4.6 Towards a linear time algorithm for general cf parsing

The algorithm of Earley avoids to do work twice. But is it possible that still too much work is done ? We remind the reader of the pattern matching problem. Too much work was done in the naive algorithm by inspecting characters more than once. In the linear time algorithms this is avoided by the introduction of the concept of failure. This happens in the algorithms of A&C, KMP, BM and Manacher.
In the algorithms for cf recognition the treatment of failure is, up till now, a neglected point. In the algorithm of Earley, and in our extended algorithms, nothing is done with the failure of items. In the algorithm of Earley common parts of grammar rules are combined, which counts for the improvement from exponential towards cubic recognition time. But failure items simply die out. This is nowhere registered. It seems to us that that kind of information can be made useful because for these failure items still an $O(n^3)$ bookkeeping has to be performed.

We did some experiments along this line of thought by making combinations of the algorithm of Manacher and our algorithm for the calculation of compatibility, but did not succeed up till now to arrive at a general improvement.

7.5 The complexity of on-line recognition and parsing of type-1 and type-0 grammars

7.5.1 Within the system Parspat

In this subsection we investigate the complexity of parsing type-1 and type-0 grammars. The complexity for transduction grammars will be the same as for type-0 grammars because the same kind of operations on the L- and R-dag are involved. Some theoretical results on the complexity of the parsing of cs grammars will therefore be relevant also for transduction grammars.
The runtime complexity will be determined by the interaction between the L- and R-dag. The interaction is registered by the datastructure of the connector. The L-dag on its own has an O(n3) runtime complexity, the R-dag an O(n) complexity because for a lhs only a FSA is created. The difference between cf- and cs-parsing is that for the latter the LTI-instruction for cs-reductions is introduced, which results in a connector in which a reference is maintained to an "active_r" : a list of active nodes in the R-dag. In worstcase situations this results in a number of lists of active-L nodes and a number of lists of active_R nodes. Within each list the $O(n^3)$ property is maintained, but the number of lists can become, for worstcase grammars, exponential.
In section 6.4.3 we discussed the use of 3 compiler switches which resolve inadequacies in the generated LR-tables. These switches cause an intervention in the case of shift/reduce-reduce conflicts, for cf- and/or cs-reduces. It is easy to implement other intervention strategies, based upon the application. In any case, when the number of type-1 or-0 reductions for a symbol in a state is kept to 0 or 1 the runtime complexity becomes less than exponential. Walters and Turnbull already showed that for deterministic type-1 and -0 parsing the runtime is proportional to the length of the parse. In the case of 0 or 1 type-1 or-0

parsing the runtime is proportional to the length of the parse. In the case of 0 or 1 type-1 or-0 reductions for a symbol in a state the runtime will therefore be, in worstcase, $O(n^3)$.llongest parsel.
We will compare our results with already existing bounds and will investigate how the runtime complexity of the PTA can be further improved.

7.5.2 Recapitulation of results about type-1 and -0 recognition and parsing

Up till now no general parsing strategies were available for type-1 and -0 grammars other than the naive technique to generate all possible sentences and to compare them with the input. We can therefore make no comparisons with other strategies.
It is fact that, in general, the recognition problem for type-0 grammars is undecidable. This can be observed directly in the PTA in the unpredictable increase and decrease of the depth of the L- and the R-dag.
The recognition problem for a csg is decidable, and Book (1978) proved that the recognition problem for a linear csg is NP-complete. He defines a linear csg as a csg which generates a sentence in linear time. The recognition time of a deterministic csg is proportional to the number of rewritings. Therefore the recognition time of deterministic linear csg's is linear.

The recognition time of an ambiguous linear csg with the system Parspat will therefore depend entirely on the complexity of recognition with a PTA. This provides us with a suitable starting point when we want to study improvements for the PTA. On the one hand we know that we are dealing with an NP-complete problem, on the other hand we may try out possible improvements until the inherent (supposed) exponential character of the problem manifests itself clearly. In order to do so we have to prepare, like we did in the cfg case, a worstcase linear csg.
As an example we show the following linear csg of Book, which is unambiguous.

<u>Source</u> : (Book, 1978)
<u>Purpose</u> : to show that recognition of linear cs-languages (to be generated in linear time from a csg) is NP-complete
<u>Result</u> : description of the CS-language LBook = { $11yxd^n$ | y ε {0, 1}* and y is the binary encoding of n, without leading 0's }
<u>Grammar</u>: GBook :

    S : 1 1 x d
    1 x : C x d
    0 x : 1 x d
    1 C : C 0
    0 C : 1 0
    l C : 1 1 0

<u>Example of generation</u> : S => 1 1 x d => 1 C x d d => 1 1 0 x d d => 1 1 1 x d d d
<u>Parspat</u> : runtime O(n)

In search for a worstcase linear csg we investigated a number of grammars from the literature. They all center around the description of languages for which it is known that no cfg exists. The two favourite ones are the languages Lanbncn = { $a^nb^nc^n$ | n >= 1 } and Lpower2 = { a^i | i is positive power of 2 }, with some variations. Turnbull (1975) studied the deterministic parsing of a sub-class of type-0 grammars and developed a method to determine if a grammar is ambiguous. He presents the following two linear csg's. With the aid of the Parspat compiler it is easy to detect ambiguity by the inspection of the generated LR-tables.

Source : (Turnbull, 1975, pp. 5-11)
Purpose : to provide for a simple ambiguous cs grammar
Result : language Lanbncn = { $a^n b^n c^n \mid n >= 1$ }
Grammar1 : ambiguous grammar GTurnbull1 :

```
        S : a S B c | a B C
        a B : a b
        b B : b b
        b C : b c
        c C : c c
        C B : B C
```

Parspat : ambiguous grammar, but the number of parses is not exponential with n
Grammar2 : non-ambiguous grammar GTurnbull2 :

```
        S : A B S c
        S : A b c
        A b : a b
        A a : a a
        B b : b b
        B A : A B
```

Parspat : non-ambiguous grammar; there is always one parse, for any n.

Because we did not obtain a linear csg with an exponential number of parses we constructed the following worstcase linear cs grammar, called "gramcss", which combines the worstcase aspects of cf and of cs grammars.

```
(1)     S : S S
(2)     S : a
(3)     a a : a a a
(4)     a a : b a a
(5)     a b : a a
```

The language which is generated by the grammar is simply a^n, with n >=2. This is not a context sensitive language, but for our purposes this is not relevant.
The grammar is inherently ambiguous because the rhs of rule (5) is a suffix of the rhs of the rules (3) and (4). The number of parses is exponential. This is shown by the following calculations.
Only those cs-reductions which lead to a string of only a's are permitted. Therefore, the 'b' which is introduced in rule (5) has to disappear by a rewriting with rule (4).
Define :
CF(n) = the number of parses for input a^n , with only cf rewritings.
CS(n) = the number of parses for input a^n , with and without cs rewritings.
$c_1(n)$ = the number of cs rewritings of a^n to a^{n-1}.
$c_2(n)$ = the number of double rewritings of a^n to a^{n-2}.

Then we have : $CF(n) = {}_{j=1}\Sigma^{n-1}\ CF(n-j).CF(j)$, which sums to $1/(n+1)\ ({}^{2n}{}_{n})$, as we saw already with grammar UBDA in section 7.4.4.2.
$c_1(n)$ = with rule 3 : (n-2), with rules 4 and 5 : (n-3), therefore in total (2n-5).
Then $CS(n) = CF(n) + c_1(n).CS(n-1) - c_2(n).CS(n-2)$.

In this formula we see that both cf and cs rules contribute to the exponential behaviour.

Input = a^n	Number of Parsetrees (CS(n))			CF(n)	c1	c2
	Calculated	Constructed by Parspat	cpu-time			
1	1	1		1	0	0
2	1	1		1	0	0
3	3	3		2	1	0
4	14	14		5	3	0
5	84	84	10s	14	5	0
6	616	616	1:11m	42	7	1
7	5256	5256	12:28m	132	9	5

7.5.3 Towards improvements of the PTA

We will indicate globally how the runtime complexity of the PTA can be improved.
1. Some instances of undecidability can be removed when all connectors are kept and a new connector is compared with older ones with regard to all information to which it refers. In such a way the repetition of an identical situation can be trapped.
2. In the analysis above we saw that in the case of type-1 and -0 grammars a multitude of active lists can be present, which contributes to the (worstcase) exponential behaviour. It is possible to develop a PTA with only one active list in the L-dag and one active list in the R-dag and to show how recognition, but not parsing, can be speeded up for e.g. the grammar gramcss. We will publish these results later.

7.6 The complexity of transduction with finite delay of transduction grammars

The complexity of transduction is the same as the complexity of parsing of type-1 and -0 grammars. The naive implementation cycles through the rhs's of all grammar rules in order to find a match. When a match is found, the rhs is replaced by the lhs, in the input. When more matches are possible usually the first one is chosen, which may be against the intention of the grammar writer. Apart from this, the runtime complexity will be dependent of the number of rules. The PTA operates for unambiguous transduction grammar in a time which is proportional to the number of rewritings, independent of the number of grammar rules.

All information which is gathered during the parsing of a rhs is, upon the reduction of that rhs, available for the lhs by simply passing to a connector the pointer to the constructed path for the rhs. When this path is used in the R-dag for the evaluation of cover symbols the path is called an association list. The evaluation of a cover symbol takes time proportional to the depth of the subtrees in the associationlist. If there are alternative subtrees the efficient representation of the former parse can be exploited to the bone: a sub-dag can be build during the evaluation of a cover symbol, in the same way as when alternatives in a text may be read.

7.7 The complexity of the introduction of tree symbols in grammars

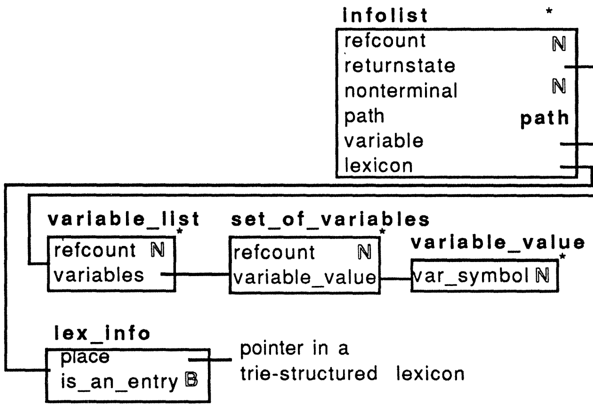
In section 6.5.2 we discussed the generation of skip instructions for grammars which contain tree-symbols. If in the input (usually a large tree-structured text file) with an opening "(" a pointer is maintained to its corresponding ")" then, after the matching of a subtree-pattern, a jump may be made in the input to that ")". In the case of type-4 pattern grammars with tree-symbols there will be, on the average, a sublinear behaviour.

7.8 The complexity of the introduction of Boolean operators in grammars

In the chapters 5 and 6 we showed how Boolean operators are treated with the compiler in the case that the switch "multi"=false (then it is assumed that no nonterminals are rewritten with infix-recursion). In runtime nothing reminds the presence of the Boolean operators. Therefore the runtime behaviour is better than that of the algorithm of Ken-CHi-Liu (1981, p. 2.31).

7.9 The complexity of the introduction of variables in grammars

ATN's and Attribute Grammars are the most frequently used grammar formalisms which make use of variables. Their implementation is usually achieved with the aid of backtracking, which accounts in worstcase for an exponential runtime behaviour. For these grammar formalisms only one symbol appears at the lhs of a rule. In order to analyze the behaviour of the PTA for the same formalism we repeat from chapter 4 the datastructure of an infolist.



In an infolist are brought together a path in the parse forest, a list of variables which their (multiple) values and a pointer in the lexicon. The number of created infolists depends upon the number of reduces which takes place and is not influenced by the number of values of variables. The number of reduces for grammars with rules with a |lhs| = 1 has an upper bound of $O(n^2)$ (the same as with Earley's algorithm). An increase of the upper bound of the runtime complexity can therefore only be attributed to a multitude of values for sets of variables. This multitude is represented in the datastructure by multiple elements of "variable_list". Each of these elements points to a "set_of_variables" in which for each variable in the grammar rule only one value is stored. After the reading of a symbol a shift or a reduce may be performed with the item to which the infolist belongs (in the poplist). The variable_list will be copied to the new infolist, with the current pointers to set_of_variables. When an assignment is made to a variable then the new variable_value for that variable is created.

Two cases can be distinguished :

1. each variable has a finite set of possible values; when in runtime a check is performed in order to detect elements of variable_list with the same set of values then the upper bound for the runtime complexity still remains $O(n^3)$ with a constant factor which is influenced by the size of the set of possible values for each variable;

2. if the set of possible values for at least one variable is not finite then the upper bound for the runtime complexity becomes exponential; it is possible that more infolists will be created for a current item. On a shift or reduce with that item and the execution of an assignment to a variable as an action the refernce to a variable may be found in multiple infolists. This accounts for a multiple number of elements in variable_list. (Our approach of creating dags in order to keep the runtime complexity polynomial fails here because no sequence is involved in the variables to which values may be assigned : at each place in a rule all variables may be assigned a value and therefore each combination of values of variables has to be kept as a separate set in variable_list.)

The behaviour of the system Parspat for case 2. is demonstrated by the following two grammars. To the variables are assigned values with a variable length. We performed some measurements on the cpu-time which is needed for recognition according to these grammars.

The first grammar for case 2. with the name Gpalindvar is the grammar for palindromes Gpalind (see section 7.4.4.4) extended with a variable in which we build the current parse, in parallel with the parse which is constructed in the parse forest. The grammar is unambiguous and reads as follows:

```
S(O:v) ::    a{v := %}, [S(w){v := v || '(' || w || ')'}], a{v := v || %} |
             b{v := %}, [S(w){v := v || '(' || w || ')'}], b{v := v || %}.
```

The measurements are :

input :	cpu-time in seconds
a^2	0.04
a^4	0.13
a^6	0.19
a^8	0.28
a^{10}	0.37
a^{50}	11.27
a^{100}	30

Conclusion : there seems to be a runtime behaviour between $O(n^2)$ and $O(n^3)$. The reason is that the grammar is not ambiguous but that temporarily more infolists are build in order to keep track of multiple returnstates. Most of these infolists, together with their values of variables, are pruned thereafter.

The grammar for case 2. with the name Gpalvar is inherent ambiguous and looks like the grammar for palstars Gpalstar (also in section 7.4.4.4). It is extended with a variable x to which we assign multiple values with a variable length.

```
S(O:x) ::     a{x := 'a'}, T(x).
T(IO:x) ::    a, T(x), T(x) |
              a{x := x || 'a'} |
              a{x := x || x}.
```

The measurements are :

input :	cpu-time in seconds	number of parses
a^4	0.2	10
a^6	0.7	16
a^8	9.6	80

Conclusion : here we demonstrate an exponential behaviour.

7.10 The complexity of the introduction of lexicon symbols in grammars

Each infolist contains only one lexicon-pointer. The instruction LEXINC increments the pointer in the lexicon upon a terminal symbol that is read in. In section 2.2.7.2 we described the trie-structure of a lexicon with which the PTA cooperates. In (Skolnik, 1982) is described how the access to a trie-structured lexicon on disc can be optimized. Provisions are taken that substrings of an entry are kept as much as possible within one disc-page. Together with a built-in cache memory handling of these pages the number of accesses to external memory can be kept to a minimum. We measured an average between 2 and 3 disc-accesses for an arbitrary entry in the Longman Dictionary, which contains about 60.000 entries.

8. Applications

In this chapter we present some applications of the Parspat system. We start with an example of an input text with a complicated structure : the representation of Hittite clay tablets. Then we will exemplify the process of recognition by the application of syntactic pattern recognition in large enriched corpora of texts, music, historical and bibliographical records. The process of parsing is represented by a subset of an apsg-grammar for simplified English and by a small transformational grammar. The process of transduction is demonstrated by a grammar for the translation of Dutch number names and by a subset of a grammar for grapheme-phoneme transformation. Finally the combination of transduction and parsing is exemplified by a grammar for compound Dutch words.
All examples are taken from projects which have run or still run in the Faculty of Arts of the University of Amsterdam in the period 1978-1987, in close collaboration with the author. The results of most of these projects have been published elsewhere. Some of them were obtained with a predecessor of the Parspat system, the "Query" program. For the sake of uniformity, we changed the formalism of "Query" into the unifying formalism in the examples concerned.

8.1 Physical codings : Hittite clay tablets

This section shows an example of a text with a complicated structure. It serves to demonstrate the numerous additions to a plain text which can be made by a philologist in order to make it a valuable resource for philological and linguistic investigations. In order to operate on such an enriched text it is necessary to describe its structure by a rigorous syntax and to transform the text with the aid of a transducer like the Parspat system. That will be dealt with in the next subsection.

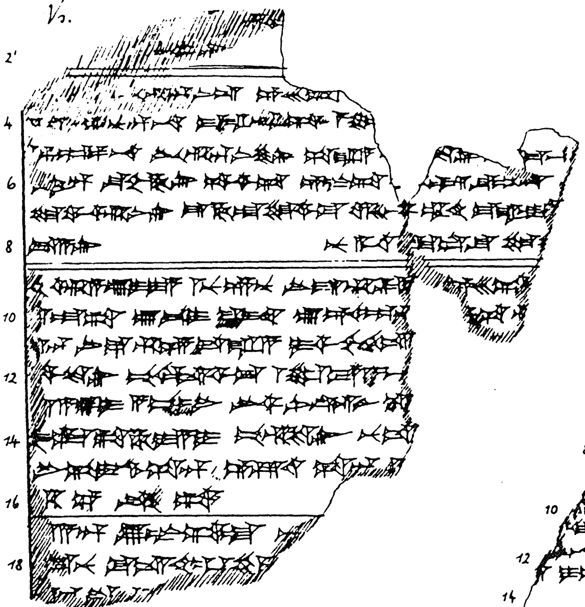
The text concerns a corpus of Old- and Middle-Hittite texts from the periods 1700-1500 and 1450-1380 BC. These texts are found on clay tablets which may be broken or damaged and which may be variants of each other. The languages used are mainly Sumeric, Accadic and Hittite. They are written on the clay tablets in cuneiform. The philologists Houwink ten Cate and De Roos (1984) added to each word in the text emendations, morphological codes, syntactic codes, lemmata (or notations to construct a lemma automatically) and sentence and paragraph classifications. In the transliteration of the text they tried to retain the original physical appearance of columns, paragraphs and lines. One of their goals was to make available a corpus suitable for syntactic pattern recognition with the Parspat system. Other goals were to construct indices, to improve automatic lemmatization and to print out the image of individual clay tablets with a complementation by other tablets containing variants of the text.

First we give an impression of the physical appearance of a broken tablet. Then we give an example of the complementation of individual tablets by other tablets. In section 8.2 we will discuss the general principle of the interactive description of a text by a grammar and the process of its transformation into a tree-structured file which is meant to be searched for by the runsystem of Parspat. The examples in this subsection serve to show that texts, in general, may take the form of a network and to show a specific operation on that network.

Nr. 104

300/f

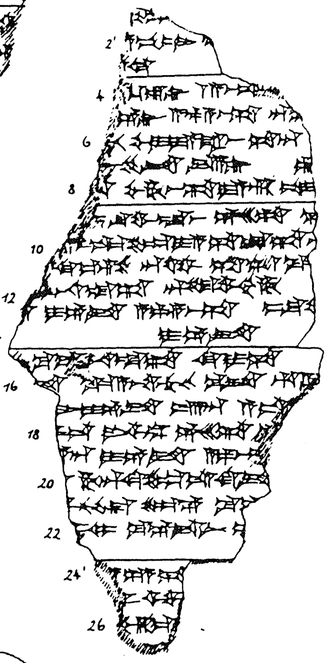
Vs.



Nr. 105

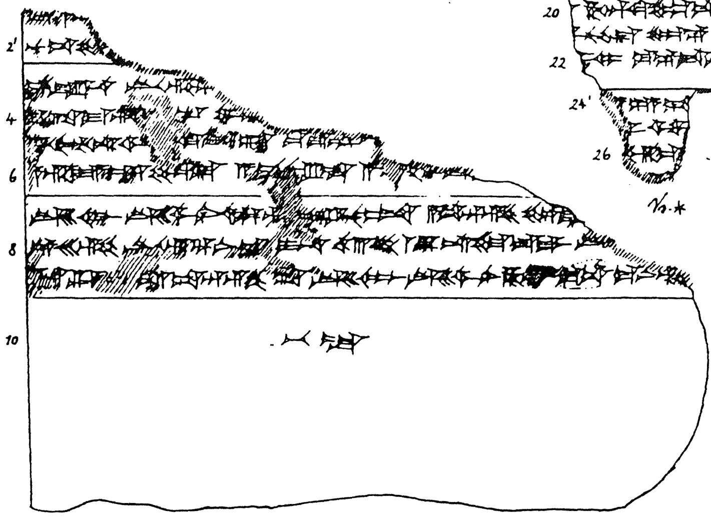
Bo 7847

Rs.



Vs.*

Rs.

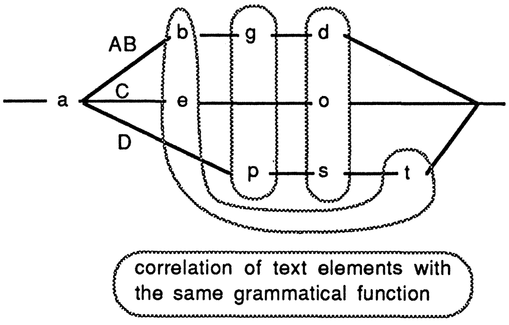


KUB XXXVI

The conventions for the coding of the clay tablets are described in (Van der Steen, Houwink ten Cate and De Roos, 1981). We only highlight the coding of parallel tablets ("variants"), which may be used for all kinds of parallel manuscripts.

As a separator for variants we use a "V", together with an indicator for the respective manuscripts, "A", "B", "C" etc.

Examples: suppose there are 5 parallel manuscripts A, B, C, D and E. An arbitrary string we denote by a, b etc.
Ex. 1 : a VA, b , AV g Manuscript A has, in addition to the other manuscripts, text b
Ex. 2 : a VA, b , AVB, g BV d Manuscripts A and B have in common texts a and d, but they differ in between a and d; A has text b, B has text g
Ex. 3 : a VAB, b , ABVC, g , CVDE, d , DEV e : Manuscripts A and B, C, D and E are, in their combinations, variants of each other
Ex. 4: VA, a , AV[b]V: the philologist (denoted by square brackets) suggests the variant b on a
Ex. 5 : a VAB, b (a) g (b) d (c) , ABVC, e (a) o (c) , CVD, p (b) s (c) t (a) , DV : the strings of the variants may be subdivided in substrings which are grammatically correlated in a different order, indicated by the characters a, b, c etc.; this is graphically represented in the next figure :


AB
C
D
a
b
g
d
e
o
p
s
t
correlation of text elements with the same grammatical function

Example of a transformation of the text based upon the variants.

We now show a text which is found on 6 different clay tablets (each broken in a different place). For simplicity we take a piece of transcribed text that only appears on 2 tablets. We call them manuscripts A and B. The output concerns the 2 manuscripts separately, where breaks are restored on the basis of the text of the other manuscript (between round brackets) or by the philologist himself (between square brackets). If the manuscripts differ in text from each other (variants) then it will be indicated by the program in a footnote. We will comment on some of the codings in the text.

```
+A3' +B2' @ [ERÍN.MEŠ]∽AB, #0BCY# Ù· #14CKB# GIŠGIGIR/VB,
I.MẸŠ,BV #0BDY# @ #0BCY# ,B∽ A-NA=A, In this last sign
KBo 22-4 3' (right) joins to KUB 40-5 II 1' (left). Line-
numbering of KUB 40-5 (II) 1' henceforward adapted. ,A=
#10CB# LÚ #0BAY# URUZa-al-pa/ #17AGA# ,A∽[(8)]]
## +A4' +B3' [p\e\$í$-r]∽AB,a-an%$n$a #10AAA# #14BAD#
MAt-ra-d,B∽u/-uš #17ADB# ḫu-u-ja-an-z,A∽[a]
{ḫuyāi-/ḫuija-} #3AAAD#
```

Comment on the 3 lines beginning with +A3' :

+A3'	For manuscript A the 3rd line from above starts. The start of the line is broken.
@	Start of a number of words that will be taken together in the index
[	The philologist supposes that the following text (which is now unreadable) appeared here
ERIN.MES	A Sumeric word.
]	End of the supposition.
~AB,	For both manuscripts A and B text becomes available
#OBCY#	The foregoing word gets the morphological classification OBCY
`*U*	Accadic word
GIŠGIGIR	Sumeric part of a word with a prefix (together 2 cuneiforms)
/	For the determination of the total lemma the whole word may be taken
VB,	Manuscript A deviates from B from this point on
MES	Suffix of the former Sumeric word
BV	End of the deviation of B. There follows no deviation for A, so B has the foregoing wordpart as an extra (will be indicated by an automatic footnote in the compound manuscript)
@	End of the already indicated number of words
, B~	Manuscript B breaks
A-NA	Accadic word
=A,	The philologist now starts a comment on manuscript A alone

A piece of the output of manuscript A, with restoration based on B, and together with footnotes from the editor and from the program looks as follows:

3′ [ERÍN.MEŠ] *Ù* $^{\text{GIŠ}}$GIGIR[8]) *A-NA*[9]) LÚ $^{\text{URU}}$*Za-al-pa* [

4′ [*pí-r*]*a-an-na* $^{\text{M}}$*At-ra-du-uš* *ḫu-u-ja-an-z*[*a*]

[8]) B II? 2′ .MEŠ ADDIDIT

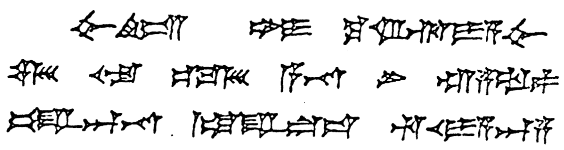
[9]) In this last sign KBo 22-4 3′ (right) joins to KUB 40-5 II 1′ (left). Line-numbering of KUB 40-5 (II) 1′ henceforward adapted.

Idem for B (no footnotes)

2′ [ERÍN.MEŠ] *Ù* $^{\text{GIŠ}}$GIGIR.MEŠ [(*A-NA* LÚ $^{\text{URU}}$*Za-al-pa*)

3′ [*pí-r*]*a-an-na* $^{\text{M}}$*At-ra-d*[(*u-uš* *ḫu-u-ja-an-z*)*a*]

The reconstructed piece of the original clay-tablet, based on the (varying and broken) clay-tablets A and B, would look as follows (printed on a Versatec printer/plotter and making use of a set of digitized cuneiform characters) :



Conclusion.

The structure of a text with variants is, in general, a dag. Therefore, this should be the general input structure for the Parspat system. The PTA itself works with the two dag-structured L- and R-stacks, of which the R-stack acts as input for the L-stack. The parse forest also has the structure of a dag. A dag structure for the input will make this picture fairly complete. However, up till now we allow only for a tree-structured input in the Parspat system. In the application of Hittite clay-tablets we therefore transform the complete text into the contributions of the individual tablets, with a possible complementation by the other tablets, as we explained in the example.
The structure of the input can be described by a grammar. With the aid of that grammar we can transform the input into a tree-structured file prepared for fast pattern matching. The process of description and transformation is described in the next section.

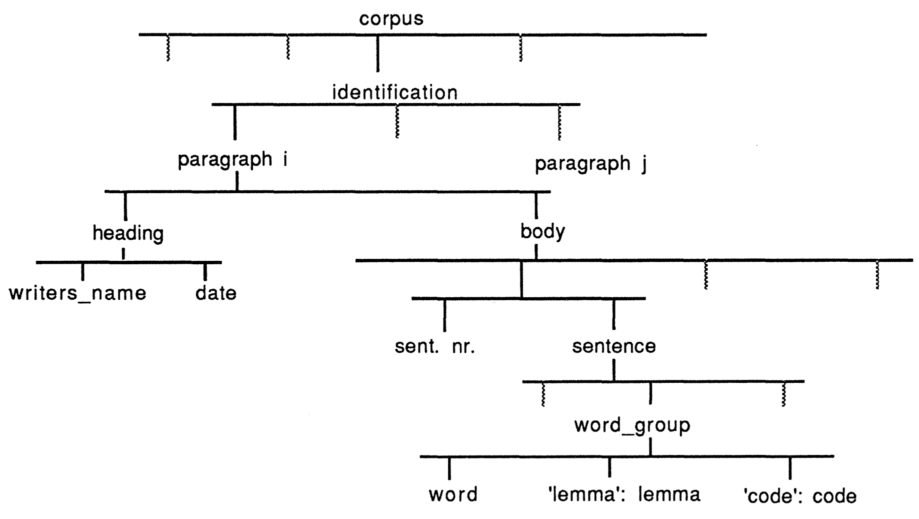
8.2 The process of the description, transformation and querying of free text

Texts are usually assembled by people who have the tendency to work methodically and to organize their texts in a top-down fashion (and if not, they can easily be persuaded to do so). It is therefore natural to describe the organization of a text by a cfg. The advantages are :
- by using a parser generator one may instantly create a parser to analyse the text in atomic parts
- the text grammar is a precise description and may as such be used in other experimental programs which make use of the text
- a text grammar provides for variable-length records
- during the methodic construction of a new text according to the grammar there may be an instantaneous syntax check on the input (if the recognizer has the on-line property, as does the Parspat system)
- the grammar reflects the hierarchical structure of the text and as such aids in the building of a tree structure.

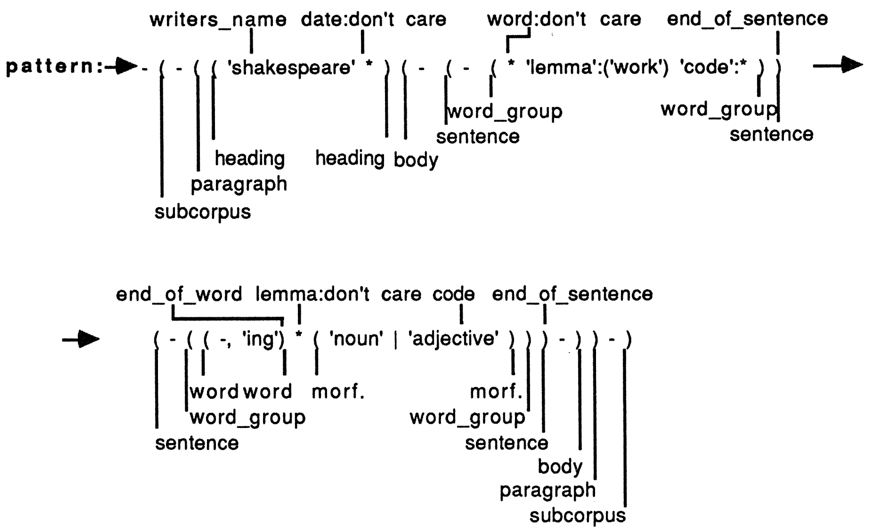
In the next figure we give an example of a simple grammar for a text-corpus.

```
corpus               :: [ subcorpus ]*.
subcorpus            :: '<', identification, '>', [ paragraph ]* .
identification       :: 'c', number.
paragraph            :: heading, body.
heading              :: writers_name, date.
body                 :: [, '<', sentence_number, '>', sentence ]*.
sentence             :: [ word_group ]*.
word_group           :: word, ' ', [ lemma, ' ' ], morphological_coding, ' '.
morphological_coding:: digit, [ char ]* .
word                 :: [ char ]+ .
lemma                :: '{', word, '}'.
         ............... etcetera .............
```

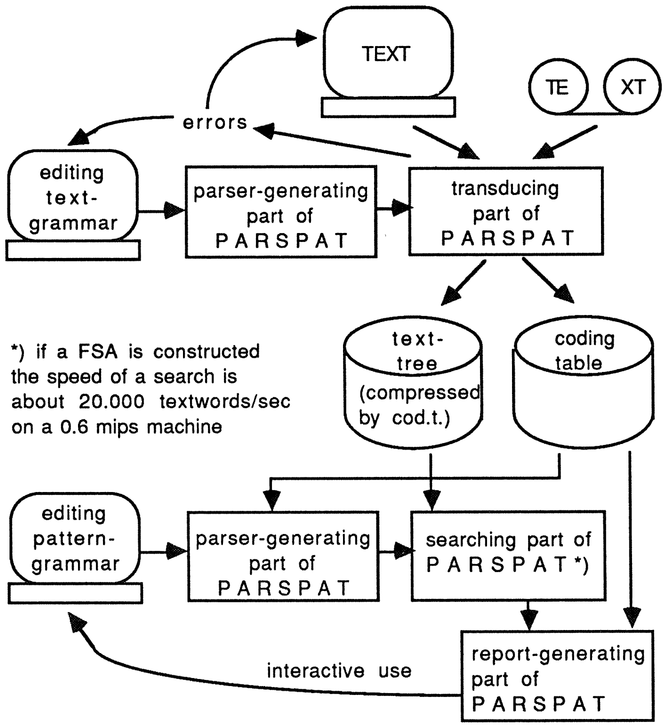
A corpus which is treated by the above corpus-grammar may be organized in the following tree-structure:



All nodes which are brothers may be seen as a consecutive piece of text on which pattern matching can take place. For that purpose we have to use the tree symbols in the query-grammar. The next pattern grammar queries for all occurrences in the work of Shakespeare where a sentence ending with a form of the lemma 'work' is followed by a sentence ending with a noun or an adjective in 'ing'.



The development of a grammar for a text for which the structure is unknown is a process of trial and error for which a program generator like Parspat is of great help. The whole process of the development of a text grammar, the transduction of the text according to that grammar and the querying of the text is depicted in the following schema.


TEXT
TE
XT
errors
editing text-grammar
parser-generating part of PARSPAT
transducing part of PARSPAT
*) if a FSA is constructed the speed of a search is about 20.000 textwords/sec on a 0.6 mips machine
text-tree (compressed by cod.t.)
coding table
editing pattern-grammar
parser-generating part of PARSPAT
searching part of PARSPAT *)
report-generating part of PARSPAT
interactive use

Nearly all texts contain errors. Therefore, after the stabilization of the text grammar errors in the text will show up. They will be reported by the recognizer generated.
In the following subsections we will give some examples of the second part of the process : the querying of text-corpora .

8.3 Pattern recognition in American, Dutch and Latin corpora

In this subsection we show a few examples of syntactic pattern recognition in corpora of English, Dutch and Latin texts with the aid of the Parspat system or one of its predecessors. For a short overview of corpus based research we refer to chapter 1.

Example Brown Corpus

<u>Remarks on the Brown Corpus.</u>
The Brown Corpus consists of samples taken from American English texts. 16 genres are covered totalling 1 million words. Each word is provided with an alphanumeric morphological code. The corpus was constructed by Francis and Kucera (1964) and is presently distributed by the Norwegian Computer Centre for the Humanities, Bergen (Norway), on behalf of ICAME, the International Computer Archive of Modern English. It is available for scientific research.

The corpus is transformed into a tree-structure on which the runtime system operates. The two most essential labels are "W" for a word in the text and "C" for a morphological code. The results of the queries are shown in the form of the sentences in the original corpus.

Example (provided by P. Masereeuw).

<u>Query</u>: Locate sentences with two adjacent negative elements, one at word-level and one below word-level.

<u>Pattern</u>:

```
S :: -, ( - C:( § ), W:( ['UN'| 'IM']1, -  ), C:( 'JJ' ), - ), -.
```

<u>Comment on the pattern</u> (if not mentioned in the former ones) :

C	:	label in text for morphological code
W	:	label in text for word
§	:	code for a negation word ("not" and "n't")
['UN'| 'IM']1	:	Between [and] we are asking for words which start with either "un" or "im"
JJ	:	the selected word has to be an adjective

<u>Part of the response</u>:

A:684 - 08 1570 but CC if CS the AT administration NN should MD find VB it PPS does DOZ not § need VB the AT $28 NNS million CD for IN a AT grant-in-aid NN program NN , , a AT <u>not § unlikely JJ</u> conclusion NN , , it PPS could MD very QL well RB seek VB a AT way NN to TO use VB the AT money NN for IN other AP purposes NNS .

A:3430 - 37 1030 reprisals NNS are BER <u>not § unheard JJ</u> of IN in IN such JJ situations NNS , , but CC the AT recent JJ tendency NN has HVZ been BEN for IN the AT Congress NP to TO forgive VB its PP$ prodigal JJ sons NNS .

Example Eindhoven Corpus

The following example (provided by J. de Jong), demonstrates the interactive querying of a corpus, in search for a particular linguistic phenomenon.

Remarks on the Eindhoven Corpus.
The Eindhoven Corpus consists of samples taken from Dutch texts. 9 genres are covered with a total of 600.000 words. Each word is provided with a numeric morphological code. The corpus was constructed in a project sponsored by the Dutch Research Council (ZWO). A description of the corpus, together with calculated frequencies of words and lemmata can be found in Uit den Boogaart (1975).

The corpus has been transformed into a tree-structure on which the runtime system operates. The two most essential labels are "W" for a word in the text and "C" for a morphological code. The results of the queries are shown in the form of the sentences in the original corpus.

Query: We want to study all sentences with a definite neutral noun-phrase containing (in departure from the general rule) an undeclined adjective or participle.

First pattern:

```
        S :: -, ( - W:( 'HET' )  C:( 3, -),  *, C:( [1 | '206' | '216']1 ), *, C:( '000' ), -), -.
```

Part of the response:

1: 43 - Het 37 vorig 1 jaar 000 kon 275 Bre!ero's 012 " De Spaanse Brabander 01 " nog 5 uitkomst 000 bieden 21 ; dit 37 jaar 000 ziet 263 men 44 zich 34 al 5 voor 6 problemen 001 gesteld 216

1:114 - De 37 psychologiestudenten 001 van 6 Amsterdam 01 , die 42 zich 34 het 37 afgelopen 206 jaar 000 als 72 " activisten 001 " deden 276 kennen 21 , laten 274 zich 34 nu 51 lelijk 15 in 6 de 37 kaart 000 kijken 2 .

1: 208 - De 37 rijstprijs 000 vormt 253 met 6 die 36 van 6 - eveneens 5 met 6 hulpgelden 001 in 62 te 65 voeren 21 - textielgoederen 001 , gedroogde 217 vis 000 , braadolie 000 , petroleum 000 en 7 een 45 paar 000 andere 453 artikelen 001 , de 37 zg. 103 prijsindex 000 van 6 het 37 dagelijks 1 levensonderhoud 000

1: 691 - Jungk 01 staat 243 kritisch 1 tegenover 6 het 37 militair-industrieel 1 complex 000

Comment on the response:
Some of the matched adjectives and particles end in "-en", like in "het ijzeren gordijn". They are uninflected for purely morphological reasons. Therefore a second pattern is created which extends the first one with a Boolean negation in order to disregard these cases.
(Such Boolean restrictions often emerge during interactive sessions.)

Second pattern:

```
        S :: -, ( - W:( 'HET' )  C:( 3, -),  W:( `[-, 'EN'] ), C:( [1 | '206' | '216']1 ), *, C:(
'000' ), -), -.
```

Comment on the pattern: the second word of the preceding pattern is now replaced by '[- , 'EN'], which indicates a word not ending in 'EN'.

Part of the response: sentences 43, 208 and 691 appear again, but sentence 114 does not.

Conclusion: After inspection of the whole output the conclusion may be justified that the adjective or participle remains undeclined in mainly the following cases (these cases are represented in the response shown):

1. when some analogy is present ("het vorig jaar", according to the adjunct of time "vorig jaar")
2. when the adjective has an adverbial function ("het dagelijks levensonderhoud")
3. when the group as a whole has a name-like function ("het militair-industrieel complex").

Example Liege Corpus

Remarks on the Liege Corpus

The Liege Corpus consists of samples taken from Latin texts. 19 samples of 8 writers have been selected, totalling 300.000 words. Each word is provided with a morphological code and a lemma. The corpus is part of a larger corpus that is distributed by "Laboratoire d'analyse statistique des langues anciennes", Liege, Belgium.

The corpus is transformed into a tree-structure on which the runtime system operates. The three most essential labels are "W" for a word in the text, "L" for a lemma and "C" for a morphological code. The results of the queries are shown in the form of the sentences in the original corpus, without codes and without lemmata.

Example (provided by J. de Jong).

Query: Give all sentences with passive verbs, accompanied by an Agent-adjunct

Pattern:

```
        S      :: -, ( -, AB, [ V ]1..4, PVF, - ), - .
        AB     :: W:( A, B ), C:( *, 7, - ).
        V      :: *, C:( `[*, 5, *, *, 1, - | *, 5, *, *, 3, - | *, 8, 2, - | *, 4, 6, - |
                  *, 4, 7, - | *, 6, 6, -]1 ).
        PVF    :: *, C:( *, 5, [A | B | C | D | E]1, - ).
```

Comment on the pattern: This pattern demonstrates the possibility of the use of nonterminals (AB, V and PVF). It is the outcome of a trial and error process with the purpose of:

- the location of as many sentences as possible
- the avoidance of "filthy" output : sentences where AB
  and the passive verb do not belong to the same syntactic unit.

The pattern describes a sentence where a word with lemma AB is followed by a passive verb. Between them 1, 2, 3 or 4 words may occur which are not a finite verb, a subordinate conjunction or a relative pronoun (the "not" is indicated by the apostrophe after the closing square bracket of <W>).

Part of the response:

A:6 - CAES, BG, 3A:1, 5 eorum una pars quam Gallos obtinere dictum est initium capit a flumine Rhodano continetur Garunna flumine Oceano finibus Belgarum attingit etiam ab Sequanis et Helvetiis flumen Rhenum vergit ad septentriones.

A:18 - CAES, BG, 3A:3, 4 in eo itinere persuadet Castico Catamantaloedis filio Sequano cuius pater regnum in Sequanis multos annos obtinuerat et ab senatu populi Romani amicus appellatus erat ut regnum in civitate sua occuparet quod pater ante habuerat item que Dumnorigi Haeduo fratri Diviciaci qui eo tempore principatum in civitate obtinebat ac maxime plebi acceptus erat ut idem conaretur persuadet ei que filiam suam in matrimonium dat.

8.4 Pattern recognition in written music : the "Cantigas de Santa Maria"

Musicologists are often interested in patterns which appear in pieces of music. As an example we mention Plenckers (1984) who investigated the hypothesis that the "Cantigas de Santa Maria" (collected in Spain in the 13th century) were influenced by the traditional Algerian song form, the "muwashshah". The method he developed was to see if certain stereotype rhythmic as well as melodic patterns were equally present in both types of music. We illustrate a transcribed piece of the "Cantigas de Santa Maria" together with its original form. Text and music (together) are in this transcribed form searched for patterns by the Parspat system.

1. a piece of the original text

[C]omo Santa Maria guareceu o moço pigureiro que levaron a Saxon, et lle fez saber o testamento das scrituras, macar nunca leera.

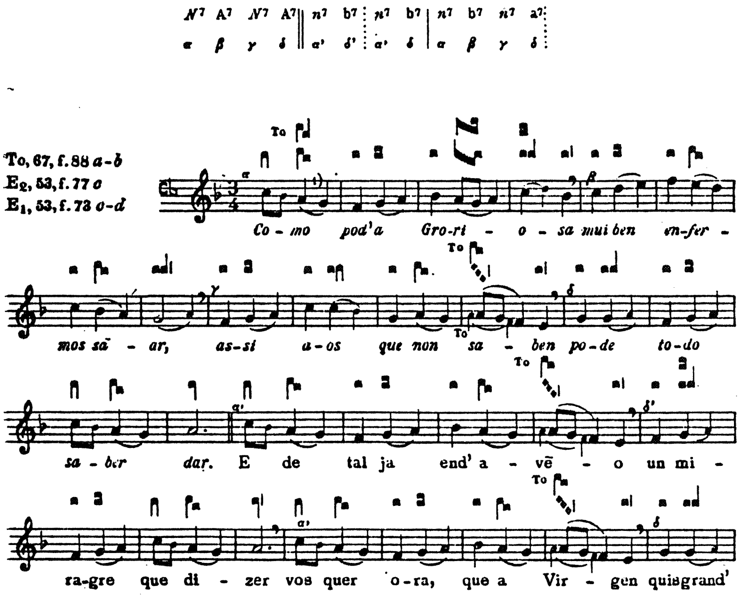

To, 67, f. 88 a-b
E2, 53, f. 77 c
E1, 53, f. 73 c-d
Co - mo pod'a Gro-ri - o - sa mui ben en-fer -
mos sã - ar, as-si a-os que non sa - ben po-de to-do
sa - ber dar. E de tal ja end' a - vẽ - o un mi -
ra-gre que di - zer vos quer o-ra, que a Vir - gen quisgrand'

Fig. 1. - *Cantigas de Santa Maria*, n. 53 - Anglés 1943-59.

2. a piece of the transcribed text (done by hand)

The following information was encoded in the data file for each song :

a) the original *Cantiga* number plus title

b) the song analysis (Roman letters concern the text structure, Greek letters the musical structure)

c) the original clef in which the song was notated

d) the text and the music of the song presented on three levels: the highest shows the original note forms; the middle level contains pitch information; the lowest level gives the text syllabically.

The way in which a, b and c have been coded can be seen by comparing the two figures. The notation of d is understood as an array of units, each containing three elements : (1) one textual syllable along with (2) the pertaining pitch information and (3) the shapes of the notational symbols. These three elements of one unit are all separated by a '/' and arranged as : shapes/pitches/syllable.

```
Cantiga 53

--------------------------

[C]omo  Santa Maria guareceu o moc*o pigureiro que levaron
a Saxon, et lle fez saber o testamento das scrituras,
macar nunca leera.

--------------------------

Alpha-N7 Beta-A7 Gamma-N7 Delta-A7=
Alpha'-n7 Delta'-b7:Alpha'-n7 Delta-b7/
Alpha-n7 Beta-b7 Gamma-n7 Delta-a7:

--------------------------

C4 Bes

--------------------------

Alpha
pq/C$b/Co- p-n/AG/mo n/F/pod' n-hnl/GA/a n/B/Gro-
bo-hn/AGA/ri- n+d/Cd/o- n'/B/sa
Beta
n/C/mui n-hnl/DE/ben n/F/en- p-n/ED/fer- n/C/mos
p-n/BA/sa~- n+d'/Ga/ar,
Gamma
n/F/as- n-hnl/GA/si n/C/a- n+pq/Cb/os n/G/que p-n/AG/non
r+r+r'/AGF/sa- n'/E/ben
Delta
n/G/po- n+d/Ga/de n/F/to- n-hnl/GA/do pq/Cb/sa-
p-n/AG/ber q/A/dar.
Alpha'
pq/Cb/E p-n/AG/de n/F/tal n-hnl/GA/ia n/B/end'
p-n/AG/a- r+r+r'/AGF/ve~- n'/E/o
Delta' n/F/un n+d/Ga/mi- n/F/ra- n-hnl/GA/gre pq/Cb/que
```

With the aid of the Parspat system the transcribed text is transformed into a tree-structured file. This file may be queried, again, with the aid of Parspat. The essential labels are "P" for "pitch" and "S" for "shape".

3. two queries :

<u>Query</u>: Search for ligatures which begin with an A or C and are followed by a rising interval.

Pattern:

```
        T :: - ( -, P:([A, - | C, -]1), S:( n, -, h ), - ), - .
```

Query: Search for a stereotyped way of cadencing at the end of a period.

Pattern:

```
        T :: - ( -, P:(-, G, - ), S:( - ), P:( A, - ), S:(-, ['n+pq'|'n+hpq']1,  - ), - ,
                P:(-, D ), S:( - ) ), -.
```

Conclusion: all the periods which were found to contain this pattern seem to show a more extensive resemblance. In all but one case, the melodic line A...D is filled in as AFED or AFEDCD; the G is consistently absent.

8.5 A historical free-text database

Recently historians have become more and more interested in the material culture of the past. Usually commercial database systems are used in order to process the material. Input to all these systems has to be in a strict format.
Sources of historical data are often in natural language. They contain a wealth of relations which are often difficult to structure in existing (textual) database systems (Hamilton et al, 1985). In that case it becomes necessary to search through all the data in a sequential way. A number of ad-hoc programs has been written to fulfil that purpose.
In (Van der Steen, 1985) and (Wijsenbeek-Olthuis, 1987) a project is described in which Delft estate-inventories are used as a source for the study of the distribution of all sorts of goods among different income-groups. These 18th century inventories are written in a certain abbreviated notation, sometimes intermixed with natural language. The inventories had to be typed in the archives of Delft. It was not known beforehand exactly what kind of information was stored in the inventories. We therefore decided that the inventories should be typed in a free format style, but in a consistent way. Afterwards the investigator described the structure of the input by a context-free grammar, assisted by the interactive use of the Parspat system. Finally the input was transformed to a tree-structure. Because it was not necessary to do pattern matching within individual words the words were coded as integers by a program. In this case the Parspat system replaces individual words in the query-grammar by their code (see also the schema in section 8.2). The words do not have to be surrounded by quotes in this case.

1. a piece of the original text, transcribed from the original in a slightly different form in order to be consistent in the order of description. This ordering functions within a line of text and determines the grammatical functioning of the separate words. The syntactic recognition process makes use of this ordering.

```
akte 3436-14
datum 1785
plaats delft
erfl nicolaas keetwijk
beroep mr.glazenmaker, verwer
x onbekend
beroepx onbekend
adres oude delft
belast 15
kind geen
leeftijd bejaard
=org
*huis
huis, woon-
*land
tuin
*effect
obligatie, frankrijk, a 1000 gulden, 2 2000g00s
obligatie, planters op de deense amerikaanse eilanden, rente 4% 1000g00s
=contant
rijder, goud 14g00s
gulden, drie-, 11 33g00s
=rg
```

*goud en zilver
schelling, scheepjes-, goud
munt, goud, vreemd

2. a piece of the transcribed text, put in the datastructure of a tree:

```
BOEDEL :( KLASSE :( D )
      GEZIN :( G )
      PERIODE :( 3 )
      SOORT :( K )
      FILE :( DG24 )
      AKTE :( 3436-14 )
      DATUM :( 1785 )
      PLAATS :( DELFT )
      ERFL :( NICOLAAS_KEETWIJK )
      BEROEP :( MR.GLAZENMAKER, VERWER )
      X :( ONBEKEND )
      BEROEPX :( ONBEKEND )
      ADRES :( OUDE_DELFT )
      BELAST :( 15 )
      KIND :( GEEN )
      LEEFTIJD :( BEJAARD )
      ORG RN :( HUIS KAT :( VR :( VO :( HUIS )
                                  PR :( WOON )
                                 )
                            )
                LAND KAT :( VR :( VO :( TUIN )
                                  )
                            )
            EFFECT KAT :( VR :( VO :( OBLIGATIE )
                                BIJZ :( FRANKRIJK )
                                AA :( 1000. GELD )
                                HOEV :( GT :( 2. )
                                        )
                              )
                          TAX :( 2000. )
                          VR :( VO :( OBLIGATIE )
                                BIJZ :( PLANTERS OP DE D.A.E )
                                RENTE :( 4. )
                              )
                          TAX :( 1000. )
                        )
                )
      CONTANT RN :( ONGEKAT KAT :( VR :( VO :( RIJDER )
                                         BIJZ :( GOUD )
                                       )
                                    TAX :( 14. )
                                    VR :( VO :( GULDEN )
                                          PR :( DRIE )
                                          HOEV :( GT :( 11. )
                                                 )
```

```
                                                  )
                                             TAX :( 33. )
                                           )
                          )
      RG RN :( GOUD_EN_ZILVER KAT :( VR :( VO :( SCHELLING )
                                            PR :( SCHEEPJES )
                                            BIJZ :( GOUD )
                                          )
                                       VR :( VO :( MUNT )
                                            BIJZ :( ZILVER )
                                            BIJZ :( VREEMD )
                                            HOEV :( GT :( 23. )
                                                       )
                                           )
```

3. two queries and their coding in the unifying formalism

Query: "Give a report for all income-classes and periods, but only for minors (SOORT=K), on the sum of occurrences of the goods 'kom', 'kan' and 'kruik', adjusted by the eventually given number of occurrences (HOEV) in the data base"

```
Pattern:    S ::         -, BOEDEL:(KLASSE:(*{R:1}), -, PERIODE:(*{R:2}),
                         SOORT:(K),  -, NEXT1, -).
            NEXT1 ::     *, RN:(-, NEXT2, -).
            NEXT2 ::     *, KAT:(-, VR:(-, NEXT3, -, [NEXT5], -){R:9}, -).
            NEXT3 ::     VO:(-, [KOM{R:3} | KAN{R:4} | KRUIK{R:5}]1).
            NEXT5 ::     HOEV:([GT{R:6}:(*{R:7}) | *{R:8}]1).
```

Query: "Give a report of all taxations of stocks (EFFECT) for people who own houses and land"

```
Pattern:    S::    -, BOEDEL:(-, ORG, RN:([A & B & C]1), -).
            A::    -, HUIS, KAT:(-), -.
            B::    -, LAND, KAT:(-), -.
            C::    -, EFFECT, KAT:(-, TAX:(*{R:1}), -), -.
```

8.6 Pattern recognition as a research tool for documentalists

In this section we show the use of syntactic pattern recognition as a tool for documentalists. The following piece of text stems from a corpus of bibliographic records with classifications from varying sources. Current research at the University of Amsterdam aims at the development of a strategy for document retrieval which makes use of a combination of classifications, even when a query for retrieval is put in terms of only one kind of classification.

1. A piece of the original text :

```
|0000
#a000054
|0010
 78603681
|0080
831122s1978    us      Wi   10000  eng
|0500
00 HV3022 .S92 1978
|0820
00 362.40483 19
@2450
30 Summary report of the national survey of transportation handicapped people
@2600
00 [Washington, D.C.] The Office [1978]
@6500
00 Physically handicapped United States Transportation Statistics
@6510
00 Social surveys United States Statistics
@7100
21 Grey Advertising Inc
@7100
11 United States Office of Transportation Planning, Management, and Demonstrations
@7450
00 National survey of transportation handicapped people
```

2. The transcribed text was put in a tree structured file.

3. We show four queries, stemming from four different "classification-views", which will all result in the same report:
(a) give all documents which have in the title of the document words which contain the strings "transport" and "handicap"
(b) give all documents which have in their Library of Congress Subject Heading words which contain the strings "transport" and "handicap"
(c) give all documents which are classified according to the Library of Congress Classification for "transport for the handicapped"
(d) give all documents which are classified according to the Dewey Decimal Classification for "transport for the handicapped"

4. The grammars for the four queries (displayed in a fashion which corresponds with the report) are respectively :
(a) S :: -, (-, (-, ' TRANSPORT', -) , ('@245', -), - & -, (-, 'HANDICAP', -), ('@245', -), -), -.
(b) **S :: -, (-, (-, ' TRANSPORT', -) , ('@65', -), - & -, (-, 'HANDICAP', -), ('@65', -), -), -.**
(c) *S :: -, (-, ('HV3022'), (-), (-), ('|050', -), -), -.*
(d) ***S :: -, (-, ('362.4', -), (-), ('|082', -), -), -.***

5. one record of the report (corresponding with the source shown above and presented with all codings, which may be eventually disregarded during reporting) is :

54 - #a000054 |0000 78603681 |0010 831122s1978 us wi 10000 eng |0080 00 *hv3022 .s92 1978 |0500* 00 ***362.40483 19 |0820*** 30 @2450 summary @2450 report @2450 of @2450 the @2450 national @2450 survey @2450 of @2450 transportation @2450 handicapped @2450 people @2450 00 @2600 [washington, @2600 d.c.] @2600 the @2600 office @2600 [1978] @2600 00 @6500 physically @6500 **handicapped @6500** united @6500 states @6500 **transportation @6500** statistics @6500 00 @6510 social @6510 surveys @6510 united @6510 states @6510 statistics @6510 21 @7100 grey @7100 advertising @7100 inc @7100 11 @7100 united @7100 states @7100 office @7100 of @7100 transportation @7100 planning @7100, management @7100, and @7100 demonstrations @7100 00 @7450 national @7450 survey @7450 of @7450 transportation @7450 handicapped @7450 people @7450

8.7 An Augmented Phrase Structure Grammar for Simplified English

In this subsection we show a piece of a grammar which describes "Simplified English", as defined by Fokker B.V. (trademark), which language is in use for the writing of technical manuals. The grammar is written by B. van der Korst by order of BSO Research. The chosen subset of the unifying formalism is suitable for the implementation of attribute grammars or apsg's ("augmented phrase structure grammars"). The grammar makes use of a lexicon. The parse which has to be constructed is not the internally constructed parse but a dependency tree which is built up in a variable.

Examples of (simple) imperative sentences, written in Simplified English :

1. install the clamp to connect the duct to the duct assembly.
2. do not use a cigarette lighter to find a gas leak.
3. apply the protective compound with a soft-bristle brush on the clean area of the fuselage skin.

Basic rules (only for imperative sentences) :

1. rule for imperative sentences :

```
IMP:: [ FADJ ]*, [ do,not ], Vrb, [ COM ], [ FADJ ]*.
```

2. rule for 'free adjuncts' :

```
FADJ:: Adv | PP | INF | ADVCL.
```

3. rule for the complement of a verb :

```
COM:: AP | NP |
     NP, [ Prt ], [ PP | INF ] |
     Prt, [ NP ], [ PP ] |
     PP | INF.
```

Additions to the basic rules :

1a. actions with variables for the construction of dependency trees
 b. parameters for the passing of subtrees

2. lexicon symbols with attributes; lexicon symbols are preceded by a '$'

3. tests with variables for the management of over-generation (congruency, sub-categorization)

4. other rules (e.g. coordination & ellipsis)

```
IMP(O:tree)::
 [ FADJ(fadj) {fadj1:=fadj1||fadj}
 ]*, $vrb(vrb,type,cmpl,prt), [ COM(com,ty,cmpl,prt) |
                          E {type='intr' & cmpl='' & prt=''}
                       ]1,
 [ FADJ(fadj) {fadj2:=fadj2||fadj} ]*
  {tree:='[E-GOV,'||vrb||fadj1||com||fadj2||']'} |
 ], 'do','not', $vrb(vrb,type,cmpl,prt), [ COM(com,type,cmpl,prt) |
                                  E {type='intr' & cmpl='' & prt=''}
                               ],
 [ FADJ(fadj) {fadj2:=fadj2||fadj} ]*
   {tree:='[E-GOV,'||do'||fadj1||',[E-INT,not],[E-INFC,'||vrb||com||']'||fadj2||']'}.

FADJ(O:fadj):: [ FADJ(fadj) {fadj1:=fadj1||fadj}
 ], 'do','not', $vrb(vrb,type,cmpl,prt), [ COM(com,type,cmpl,prt) |
                                  E {type='intr' & cmpl='' & prt=''}
                               ],
 [ FADJ(fadj) {fadj2:=fadj2||fadj} ]*
   {tree:='[E-GOV,'||do'||fadj1||',[E-INT,not],[E-INFC,'||vrb||com||']'||fadj2||']'}.

FADJ(O:fadj)::
 [ $adv(fadj) |
   PP(prp,fadj) |
   INF(fadj) |
   ADVCL(fadj)
  ]1 {fadj:=',[E-CIRC,'||fadj||']'}.

COM(O:com,I:type,I:cmpl,I:prt)::
  [ AP(pred) | NP(pred) ]1 {type='kww' & com:=',[E-PRED,'||pred||']'}|
  NP(obj) {type='trans' & com:=',[E-OBJ,'||obj||']'},
   [ $part(p) {p=prt & com:=com||',[E-P,'||p||']'}
   ], [ PP(prp,pp) {prp=cmpl & com:=com||',[E-PREC,'||pp||']'} |
        PP(prp,pp) {cmpl='advc' & com:=com||'[E-ADVC,'||pp||']'} |
        INF(inf)   {cmpl='infc' & com:=com||',[E-INFC,'||pp||']'} ] |
  $prt(p) {p=prt & com:=',[E-P,'||p||']'},
   [ NP(obj) {type='trans' & com:=com||'[E-OBJ,'||obj||']'}
   ], [ PP(prp,pp) {prp=cmpl & com:=com||',[E-PREC,'||pp||']'} ] |
  PP(prp,pp) {prp=cmpl & com:=',[E-PREC,'||pp||']'} |
  PP(prp,pp) {cmpl='advc' & com:=',[E-ADVC,'||pp||']'} |
  INF(inf)   {cmpl='infc' & com:=',[E-INFC,'||inf||']'}.

E::.
```

A few words in the lexicon *:*

(the values for variables are written after each word; some words have more values for variables)

```
compress#$vrb#1#into
compress#$vrb#3#into
disconnect#$vrb#1#from
disconnect#$vrb#3#from
do#$vrb#1#
does#$vrb#1#
writes#$vrb#3#no
often#$adv
```

8.8 A Transformational Grammar

In this section we discuss the application of the testing of a piece of a grammatical theory. The grammar is written according to some principles of Transformational Generative Grammar (TGG) and describes the movement of the verb particle in English. The first nine lines can be regarded as a package which is used here in the description of particle movement (line 10-16) but which may also serve as a starting point for a different grammar.
A verb may have one or more arguments (line 4) which are realized by a noun phrase, preposition phrase or a verb phrase. If one of the arguments is a subclause (infl, line 4 last alternative, also possible via line 6 and 8), the same restrictions apply to the verb in this clause as to the verb in the main clause (line 4, first 4 alternatives).
The particle is connected with the verb (sentence 1 below). It may change position but the possibilities are restricted. The particle may move across the first noun phrase (line 14, sentence 2 below). Furthermore, if two noun phrases or a noun phrase and a prepositional phrase follow each other, the particle is placed in between (line 15, sentence 3 below).
Movement of the particle is always clause internal; it can never cross a comp, which is the marker of the beginning of the clause (line 4, 5, 6; that-clause, relative clause, prepositional clause).
Sentences 7, 8, 9, 10 below are not accepted by the grammar. The particle may not skip more than one argument (sentence 7, 8 below) or a clause (sentence 9 below) or end up in a different clause (sentence 10 below, particle from the main clause in a relative clause).

During parsing with the Parspat system the user may inspect at any moment the (partial) parses which are constructed. Parses may be printed out in a tree-format. This is shown in the "example of an analysis".

Grammar (written by B. Molenaar) :

```
1. infl2              ::    infl1.
2. infl1              ::    v2.
3. v2                 ::    [n2], v1.
4. v1                 ::    v, n2, p2 | v, p2 | v, n2 | v, n2, n2 | v, comp, infl1.
5. n2                 ::    [$det], n1.
6. n1                 ::    [$adj]*, $n | [$adj]*, $n, comp, infl1.
7. p2                 ::    p1.
8. p1                 ::    $p, n2.
9. comp               ::    $plusw.
10. v                 ::    $vs | vpar.
11. vpar              ::    $v, $par.
14. n2, vpar, n2      ::    [$det], [$adj]*, $n, $v, [$det], [$adj]*, $n, $par.
15. n2, vpar, n2, n2  ::    [$det], [$adj]*, $n, $v, [$det], [$adj]*, $n, $par, [$p],
                              [$det], [$adj]*, $n.
16. comp, infl1       ::    $plusw, $v, [$det], [$adj]*, $n, $par.
```

Correct sentences are :

1. the happy dean brought IN the suspect.
2. the happy dean brought the suspect IN.
3. the happy dean brought the suspect IN on the bicycle.
4. the happy dean called up that he brought the suspect IN.
5. the happy dean called IN the suspect who put some fortune BY.

6. the happy dean called IN on the suspect who put some fortune BY.

Incorrect sentences are :

7. the happy dean brought the suspect on the bicycle IN.
8. the happy dean brought the suspect the bicycle IN.
9. the happy dean called that he saw the suspect UP.
10. the happy dean brought the suspect IN who put some fortune BY.

Example of an analysis:

before movement:

"the happy dean called IN the suspect who put BY some fortune"

```
                      infl2:
                         |
                      infl1:
                         |
     ___________________v2:______
     |                          |
 ___n2:___     _______________v1:___
 |      |      |                    |
$det  _n1:_    v:     ____________n2:__
       |  |    |      |               |
     $adj $n _vpar:  $det ____________n1:___
              |   |        |    |           |
             $v $par      $n  comp:      infl1:
                                |           |
                              $plusw       v2:
                                            |
                                       ____v1:___
                                       |        |
                                       v:      n2:
                                       |        |
                                     _vpar:   _n1:_
                                     |   |     |  |
                                     $v $par $adj $n
```

after movement:
(movement can only take place in the sub clause (by); the particle in the main clause (in) cannot be placed in the sub clause)

"the happy dean called in the suspect who put some fortune by"

```
                                 infl2:
                                    |
                                 infl1:
                                    |
     ______________________________v2:_____
     |                                    |
 ___n2:__       __________________________v1:___
 |      |       |                                |
$det  _n1:_     v:      ______________________n2:__
      |   |     |      |                         |
     $adj $n  _vpar:  $det                      (1)
              |   |
             $v $par

          _________________(1)_n1:______________________
          |            |   |   |   |    |              |
          $n  comp:[$plusw $v $adj $n  $par]CsRule:16 infl1
```

8.9 Parallel transduction rules for the translation of numbers into Dutch number names

In this section we demonstrate the use of the transduction capability of the Parspat system by the application of the translation of numbers into Dutch number names.

The characteristic of the following grammar is that, in contrast to other number grammars, all the rules may be applied in an arbitrary sequence (all rules are "waiting", at any moment, to be applied). The grammar was partly supplied by M. Elstrodt.

Grammar :

```
! thousands, millions, billions !
1    VOORSTUK^ , d , u , i , z , e , n , d , DRIETAL^ , #        ::     VOORSTUK ,
                                                                  DRIETAL , # .
2    VOORSTUK^ , m , i , l , j , o , e , n , ZESTAL^ , #         ::     VOORSTUK ,
                                                                  ZESTAL , # .
2a   VOORSTUK^ , m , i , l , j , a , r , d , NEGENTAL^ , #       ::     VOORSTUK ,
                                                                  NEGENTAL , # .

! hundreds !
3    h , o , n , d , e , r , d , C01^ , C02^ , NOTC^         ::     1 , C01 , C02 , NOTC .
                                 ! in Dutch one does not say 'onehundred' !
4    C2^ , h , o , n , d  , e , r , d , C01^ , C02^ , NOTC^ ::     C2 , C01 , C02 , NOTC .

! tens !
5    C3^ , t , i , e , n , NOTC^               ::     1 , C3 , NOTC .
6    C1^ , e , n , C2^ , t , i , g , NOTC^     ::     C2 , C1 , NOTC .
7    C2^ , t , i , g , NOTC^                   ::     C2 , 0 , NOTC .

! elimination of leading 0's !
8    NOTC^                     <  0 , 0 , NOTC.
9    C1^ , NOTC^               :: 0 , C1 , NOTC .
10   C01^ , C02^ , NOTC^       :: 0 , C01 , C02 , NOTC .

! digits !
11   e , e , n , NOTC^         :: 1 , NOTC .
12   t , w , i , n , t         :: 2 , t .
13   t , w , e , e , NOTt^     :: 2 , NOTt .
14   d , e , r , t             :: 3 , t .
15   d , r , i , e , NOTt^     :: 3 , NOTt .
16   v , e , e , r , t         :: 4 , t .
17   v , i , e , r , NOTt^     :: 4 , NOTt .
18   v , i , j , f , NOTC^     :: 5 , NOTC .
19   z , e , s , NOTC^         :: 6 , NOTC .
20   z , e , v , e , n , NOTC^ :: 7 , NOTC .
21   a , c , h , t , NOTt^     :: 8 , NOTt .
22   t , a , c , h , t         :: 8 , t .
23   n , e , g , e , n , NOTC^ :: 9 , NOTC .
24   t , i , e , n , NOTC^     :: 1 , 0 , NOTC .
25   e , l , f , NOTC^         :: 1 , 1 , NOTC .
26   t , w , a , a , l , f , NOTC^  :: 1 , 2 , NOTC .
```

Rewriting of the nonterminal symbols :

```
C01        ::  C0 .
C02        ::  C0 .
DRIETAL    ::  C0 , C0 , C0 .
ZESTAL     ::  DRIETAL , DRIETAL .
NEGENTAL   ::  DRIETAL , ZESTAL .
VOORSTUK   ::  C1 , [C0 , [C0]] .
C0         ::  0 .. 9 .
C1         ::  1 .. 9 .
C2         ::  2 .. 9 .
C3         ::  3 .. 9 .
NOTC       ::  0 .. 9' .                    ! the negation of a digit 0 .. 9 !
NOTt       ::  t' {C:1} | 0 .. 9' {C:1} .   ! the {C:1} establishes a
                                                    Boolean 'and' !
```

<u>Input.</u>

The structure of the input is : [1..9][0..9]* (to a maximum of 12) followed by the end-marker #.
(The grammar may be extended in a trivial way for larger numbers.)

<u>Output.</u>

Example : if the input is 2972632# then the output is :
'tweemiljoennegenhonderdtweeenzeventigduizendzeshonderdtweeendertig#'

8.10 Parallel transduction rules for the translation of graphemes to phonemes

In this section we show the use of the Parspat system for the application of grapheme-phoneme conversion for the Dutch language. The grammar consists of 9 sub-grammars which are cascaded.
The grammar was written by M. Elstrodt, together with E. Berend.

We show a piece of a sub-grammar which treats the character 'c'. First we give the rules written in standard phonological notation and then their rewriting in the unifying formalism.

Grammar written in phonological notation :

(- alternatives are denoted vertically between curly brackets; e.g.

```
        {e,  '({m,  *})) }  Means: e followed by not m and not r, u.
             {r,  u}
```

- negations are denoted by a quote
- transduction is denoted from left to right
- phonemes are written in uppercase
- graphemes in lowercase)

```
    c1:        c -> S / _ {e}
                          {i}
                          {y}

    c2:        c -> K / _ '({h})
                            {e}
                            {i}
                            {y}

    c3:        c, h -> X

    ch:        c, h -> S~ / <-segm> _ {e,  '({m,  *})) }
                                           {r,  u}
! Means: e followed by not m and not r, u.!
                                      {o,  {c}       }
                                           {w}
                                      {i,  a'        }
                                      {a,  '({k,  *})) }
                                           {o,  *}
                                           {r,  i}
                                           {r,  y}

    a,u:       a, u -> AU

    au:        a, u -> O: / S~ _
```

The grammar rewritten in the unifying formalism, as currently implemented in the Parspat system:

(- transduction is denoted from right to left)

```
! a, u !
a, @, u, AU   < a, u.                          ! the @ is a dummy which serves as an
                                            indication that the au is rewritten !

! au !
S~, a, @, u, O        <      S~, a, @, u, AU.

! c1 !
c, S, Rc1      <      c, Rc1.
Rc1            ::     e | i | y.

! c2 !
c, K, Rc2      <      c, Rc2.
Rc2            ::     '[e | i | h | y].

! c3 !
c, @, h, X     <      c, h.

! ch !
segM^, c, @, h, S~, Rch2^ <      segM, c, @, h, X, Rch2.
Rch2           ::     a, [foneem], aPAT | e, [foneem], ePAT | i, [foneem], a' | o, [foneem],
                             [c | w]1.
aPAT           ::     '[k | o | r] | r, [foneem], '[i | y].
ePAT           ::     '[m | r] | r, [foneem], u'.

! definities !
foneem         ::     @ | vowfon | confon.
vowfon         ::     AH | EH | IH | OH | OE | SW | A | E | I | O | Y | U | EU | E~ | AU | NY |
EI.
confon         ::     P | B | T | D | K | G | F | V | S | Z | X | Q | S~ | Z~ | C | L | R | W | J | H |
M |
                             M~ | N | N~ | 9.
segM           ::     grens.
universe       ::     a..z | # | &.     ! universe for the negation of phonemes !
```

8.11 A grammar for compound Dutch words

The following grammar was written (by G.J. van Schaaik) to analyse compound words of Dutch:

```
1  invoer :: comp.
2  comp :: [ N(last),  [ [cons{last=%}],  T ] ]+.
3  N(O:l) :: lexword{l:=%}.
4  lexword :: $n |  $na,  [B].
5  T :: e,  [n] |  s.
6  B :: 'er'.
7  cons :: b | d | f | g | k | l | m | n | p | r | s | t.
8  s, T <  z, T.
9  f, T <  v, T.
```

The effect of each rule is discussed on the basis of some examples. Rule 1 states that the input exists of a compound word. Rule 2 is basicly to be interpreted as follows:

(1) comp :: [N, [T]]+.

meaning that a compound word is to be analysed as an N, possibly followed by T. This sequence occurs 1 or more times, indicated by +. Rule 5 spells out the terminal symbols of T, the connectives 'e', 'en', and 's'. This tiny part of grammar accepts in principle the following types of compound words (the hyphen indicates the morpheme boundaries):

(2)	a	ei-dooier	'egg yolk'
	b	koek-e-bakker	'pastry cook'
	c	boek-en-plank	'book shelf'
	d	geluid-s-golf	'sound wave'

The symbol N is rewritten into the symbol lexword, which in its turn is rewritten into either $n or $na, the latter possibly followed by B. The symbol $n stands for a dictionary item categorized as noun, and $na means that the corresponding dictionary item noun may be followed in a compound word by an archaic connective 'er', symbolized by B and rewritten in Rule 6. The alternative at the right hand side of Rule 4, together with Rule 6 account for a correct analysis of compound words like:

(3)	a	rund-er-markt	'cow market'
	b	kind-er-feest	'child party'
	c	ei-er-markt	'egg market'

There remain two more types of compound words which can be analysed by the grammar given above. The first type constitutes a class of compounds the first noun of which is sensitive to voicing of the last consonant if it is followed by a vowel. This can be exemplified by:

(4)	a	huis + markt --> huiz-en-markt	'house market'
	b	duif + hok --> duiv-en-hok	'pigeon house'

Rule 8 and Rule 9 take care of the transduction of 's before T' --> z and of 'v before T' --> f. It implies that even erroneously spelled words like 'huisenmarkt', 'duifenhok', and ab-

berations like 'huissmarkt' and 'duifshok' will be accepted by the grammar, although it is very unlikely that words like these occur in a written text input for analysis.
The last type of phenomena that can be tackled by the grammar is the doubling of a consonant in words that have a 'short' vowel. The rules of Dutch spelling require that when the 'shortness' of a vowel is preserved in a derivation, it is reflected by doubling the final consonant. So we have for instance:

(5)	a	kip + soep	--> kip-p-e-soep	'chicken soup'
	b	kat + bak	--> kat-t-e-bak	'cat's box'

Let's consider Rule 2 in its original form together with Rule 3:

```
(6) Rule 2   comp :: [ N(last),  [ [cons{last=%}],  T ] ]+.
    Rule 3   N(O:l) :: lexword{l:=%}.
```

In (6) it is shown that the last read letter (indicated by '%') of the input is assigned to the variable 'l' and forwarded by means of Rule 3. In Rule 2 it is evaluated if the last read letter equals a consonant, that is, an element of the set rewritten in Rule 7. If the test in Rule 2 succeeds, then the connective T can be accepted.
The grammar produces an output with labeled brackets (7) which can be transformed into a tree structure (8). The output of (3b) will be given:

```
(7)  < invoer:(comp):(N:(lexword:($na:(k i n d)
                                      B:(er)
                                    )
                             )
                           N:($n:(f e e s t)
                             )
                           )
```

```
(8)            invoer
                 |
            --- comp ---
             |        |
             |        |
           lexword    N
           |    |     |
          $na   B    $n

          kind  er  feest
```

<u>Some entries in the lexicon are :</u>

afdruk#$n
mechanisme#$n
afstand#$n
bediening#$n
alarm#$n
centrale#$n
antwoord#$n
kaart#$n

nummer#$n
bad#$n
hand#$n
doek#$n
basis#$n
verzekering#$n
beest#$n
spul#$n
beroep#$n
voorlichting#$n
keuze#$n
bescherm#$n
kap#$n
bijzet#$n
tafel#$n
bivak#$n
muts#$n
bloed#$n
doorstroming#$n
boer#$n
traditie#$n
boodschappen#$n
wagen#$n
buffet#$n
kast#$n
rijtuig#$n

9. Suggestions for further research

We suggest a number of possible extensions of the research which we described in this book.

In chapter 1 we motivated the development of a program generator by the minimization of the number of transformations that have to be made by the human in order to translate a problem into a program. In chapter 2 we made an inventory of the most frequently used constructs in computational linguistics and unified them into one formalism. It remains to be seen if less frequently used constructs can also be expressed easily in the unifying formalism. On the one hand one may expect a convergence of the number of basic constructs which are in use in a discipline, on the other hand a divergence may be expected when other disciplines are used as a support. In that case new sub-formalisms will have to be added to the unifying formalism and the program-generation techniques will have to be extended.

The transduction formalisms which we presented in chapter 2 are sufficient for a number of applications. It is possible that other applications will profit from more shorthands for compact notations. In order to handle probabilistic grammars it will be necessary to allow for real arithmetic with the variables.

In chapter 4 a number of improvements suggest themselves. The compiler and the PTA can be extended in order to work with lookahead, which will decrease the runtime complexity, but which will increase the compilation time. The methods for disambiguation by the user in inadequate states can be extended. This has to be tried out in new applications. We already experimented with notations and implementations for indicating the ordering of some rules within one grammar.
The cubic runtime complexity of the recognition of context-free grammars is caused by the fact that items in nodes are shared, as is the case with Earley's algorithm. This principle can be extended further to the whole PTA, from which the treatment of variables will benefit: to share records where this is possible.
Unification of variables, like in Prolog, has to be implemented for cascaded grammars as a whole.
Error detection and -correction is an important topic. Recent research on error-recovery in LR-parsing can be used in order to include this capability in the Parspat system.
We discussed already the possibilities for implementation on parallel hardware. Intriguing is the thought of microprogramming the instructions for the PTA or, eventually, of building a complete computer upon its concept.

The runtime complexity of the compiler is an important subject. Especially during the development of a grammar the compiler is frequently called. The complexity is directly related to the number of generated itemsets. In chapter 6 we made some observations on this number. More research is needed in order to decrease it by further optimizations or different algorithms. At this time we are implementing "lazy compilation": an itemset is only created when the runsystem asks for it.
We discussed the compiler switch "multi". With the value "false" a recursive grammar is transformed into a FSA by the execution of a cf reduction in compile time. In that case the L-dag of the PTA will not be used. In the same way a process may be imagined for the avoidance of the R-dag when a cs reduction will be processed in compile time. It will then be pos-

sible to create a FSA for a sub-class of type-0 grammars which generate a regular language (which will call for a precise definition of that sub-class). In the case of decidable transduction grammars without recursion a FSA with output will then be produced. This will account for a very fast transducer in the case of, for instance, the shown application of grapheme-phoneme conversion in chapter 8.

In chapter 7 we indicated further research on reducing the runtime complexity of cfg's and linear csg's by a possible extension of the PTA.

The unifying formalism has a large potential for its use in applications, as we showed in chapter 8. We should like to explore the possibilities for its use further in general rewriting systems. Rule-based expert systems could profit from the compilation technique and of the integrated use of a lexicon in which large amounts of facts can be stored.

In general, improvements of the complexity of the compiler and the PTA obtained in the context of one application directly improves the performance of other applications. This was demonstrated in the years in which we experimented with implementations of the Parspat system.

10. Literature

Aarts, J. and Van den Heuvel, T. (1984), "Linguistic and Computational Aspects of Corpus Research". In: Aarts and Meijs, eds. (1984), pp. 83-94.

Aarts, J. and Meijs, W., eds. (1984). Corpus Linguistics, Rodopi, Amsterdam.

Aho, A.V. and Corasick, M.J. (1975), "Efficient String Matching: An Aid to Bibliographic Search", CACM, June 1975, Vol. 18, nr. 6, pp. 333-340.

Aho, A.V., Hopcroft, J.E. and Ullman, J.D. (1974). The Design and Analysis of Computer Algorithms, Addison-Wesley.

Aho, A.V. and Johnson, S.C. (1974), "LR parsing", Computing Surveys 6 (2), pp. 99-124.

Aho, A.V., Sethi, R. and Ullman, J.D. (1986). Compilers, Addison Wesley.

Aho, A.V. and Ullman, J.D. (1972). The Theory of Parsing, Translation and Compiling (2 vols.). Prentice-Hall.

Aho, A.V. and Ullman, J.D. (1977). Principles of Compiler Design, Addison Wesley.

Akkerman, E., Masereeuw, P. and Meijs, W.J. (1985). "Designing a computerized lexicon for linguistic purposes: Ascot report nr. 1". Rodopi, Amsterdam.

Anderson, S.O. and Backhouse, R.C. (1981), "Locally Least-Cost Error Recovery in Earley's Algorithm", ACM Trans. on Progr. Lang. and Systems, vol. 3, nr. 3, july 1981, pp. 318-347.

Apostolico, A. and Giancarlo, R. (1986), "The Boyer-Moore-Galil string searching strategies revisited", Siam J. Comput., Vol. 15, Nr. 1, Feb 1986, pp. 98-105.

Atwell, E.S. (1982). "Lob Corpus Tagging Project: Manual Postedit Handbook". Department of Linguistics and Modern English Language and the Department of Computer Studies, Univ. of Lancaster.

Atwell, E.S. (1983), "Constituent-Likelihood Grammar". In: Newsletter of the International Computer Archive of Modern English (ICAME NEWS), 7, pp. 34-66.

Atwell, E., Leech, G. and Garside, R. (1984), "Analysis of the Lob Corpus: Progress and Prospects". In: Aarts and Meijs, eds. (1984), pp. 41-52.

Bailey, R.W., ed. (1982). Computing in the Humanities, North Holland.

Bara, B.G. and Guida, G. (eds.) (1984). Computational models of natural language processing, North Holland.

Bates, M. (1978), "The Theory and Practice of Augmented Transition Network Grammars". In: Bolc, L., ed., "Natural Language Communication with Computers", Lecture Notes in Computer Science, Vol. 63, 1978, Springer Verlag.

Bolc, L., ed. (1983). The Design of Interpreters, Compilers, and Editors for Augmented Transition Networks, Springer Verlag.

Book, R.V. (1978), "On the complexity of Formal Grammars", Acta Informatica 9, pp. 171-181.

Bouckaert, M., Pirotte, A. and Snelling, M. (1975), "Efficient Parsing Algorithms for General Context-free Parsers", Information Sciences 9, pp. 1-26.

Boyer, R.S. and Moore, J.S. (1977), "A Fast String Searching Algorithm, CACM, Vol. 20, nr. 10, pp. 762-772.

Bullen, R.H. and Millen, J.K. (1972), "Microtext - The Design of a microprogrammed finite state search machine for full-text retrieval". Fall Joint Computer Conference.

Burgess, C. and Laurence, J., "An Indexed Bibliography for LR Grammars and Parsers". Continually updated and available by Dr. C.J. Burgess, Dept. of Comp. Sc., School of Math., Univ. of Bristol, University Walk, Bristol, BS8 1TW, England.

Buttelmann, H.W. (1975), "On the syntactic structure of unrestricted grammars", Information and Control 29, pp. 29-101.

Chaucé, J. (1974). Transducteurs & Arborescences. Doctoral Dissertation, Grenoble.

Chiang, Y.T. and Fu, S.K. (1984), "Parallel Parsing Algorithms and VLSI Implementations for Syntactic pattern recognition", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-6, nr. 3, May 1984, pp. 302-314.

Chomsky, N. (1965). Aspects of the Theory of Syntax, M.I.T. Press, Cambridge, Massachusets.

Clocksin, W.F. and Mellish, C.S. (1981). Programming in Prolog, Springer Verlag.

Cohen, B.L. (1977), "A Powerful and Efficient Structural Pattern Recognition System", Artificial Intelligence 9, pp. 223-255.

Cohen, J. (1985), "Describing Prolog by its Interpretation and Compilation", CACM, Vol. 28, nr. 12, pp. 1311-1324.

Cohen, J. and Kolodner, S. (1985), "Estimating the Speedup in Parallel Parsing", IEEE Transactions on Software Engineering, Vol. SE-11, Nr. 1, Jan. 1985.

Courcelle, B. and Franchi-Zannettacci, P. (1982), "Attribute grammars and recursive program schemes", Theor. Comput. Sci. 17, pp. 163-191 and 235-257.

Dahl, V. (1985), "Hiding complexity from the Casual Writer of Parsers". In: (Dahl and Saint-Dizier, 1985), pp. 1-19.

Dahl, V. and Saint-Dizier, P. (eds.) (1985). Natural Language Understanding and Logic Programming. North Holland.

Date, C.J. (1981). An Introduction to Database Systems. Addison-Wesley.

De Jong, J., and Masereeuw, P. (1985), "Using a Latin Computer Corpus for Linguistic Research", to appear in "Melanges Delattes", 12 pp.

De Jong, J., and Masereeuw, P. (1987), "Parscot, a new implementation of the LSP-grammar". In: Meijs, ed. (1987), "Corpus Linguistics and Beyond", Rodopi, Amsterdam, pp. 195-206.

De Mori, R. (1983). Computer Models of Speech Using Fuzzy Algorithms. Plenum Press, New York.

DeRemer, F.L. (1969). Practical Translators for LR(k) languages, Ph.D. dissertation, MIT, Cambridge, Mass.

DeRemer, F.L. (1971), "Simple LR(k) grammars, CACM, Vol. 14, nr. 7, pp. 453-460.

De Roos, H. (1984), Amsterdam. Doctoral Dissertation, University of Amsterdam.

Dershowitz, N. (1985), "Computing with Rewrite Systems", Information and Control 65, pp. 122-157.

Earley, J., 1970, "An efficient context-free parsing algorithm", CACM, Vol. 13, nr. 2, pp. 94-102.

Eeg-Olofsson, M. and Svartvik, J. (1984), "Four-level Tagging of Spoken English". In: Aarts and Meijs, eds. (1984), pp. 53-64.

Ehrig, H., Nagl, M. and Rosenberg, G. (1983). Graph-grammars and their application to Computer Science, Springer Verlag.

Elstrodt, M., Honig, J., Masereeuw, P., Portier, J., Schwartzenberg, G., Skolnik, J., Van der Steen, G.J. and Van Halteren, H. (1984). Ystrings, a package for the manipulation of strings in standard Pascal. Technical report, available from the Computer Department Faculty of Arts, University of Amsterdam.

Engels, L.K. (1981). Leuven English Teaching Vocabulary-List, Based on Objective Frequency Combined with Subjective Word Selection. Dept. of Linguistics, Univ. of Leuven.

Faloutsos, C. (1985), "Access Methods for Text", Computing Surveys, Vol. 17, Nr. 1, March 1985, pp. 49-74.

Finin, T.W. (1983), "The Planes Interpreter and Compiler for Augmented Transition Network Grammars". In: Bolc, ed. (1983), pp. 1-69.

Fischer, C.N. (1975), "On parsing context-free languages in parallel environments", Ph.D. dissertation, Dep. Comput. Sci., Cornell Univ., Ithaca, NY, Apr. 1975.

Fischer, P.C. and Paterson, M.S. (1974), "String-matching and other Products". In: Complexity of Computation (SIAM-AMS Proceedings, vol. 7), R.M. , ed., American Mathematical Society, Providence, RI, pp. 113-125.

Fisher, A.J. (1985), "Practical LL(1)-Based Parsing of Van Wijngaarden Grammars", Acta Informatica 21, pp. 559-584.

Francis, W.N. (1982), "Brown Corpus Bibliography". In: Johansson, ed. (1982), appendix.

Francis, W.N. and Kucera, H. (1964)."Manual of Information to Accompany a Standard Sample of Present-Day Edited American English, for Use with Digital Computers", Department of Linguistics, Brown University, rev. eds. 1971 and 1979.

Francis, W.N. and Kucera, H. (1982). Frequency Analysis of English Usage: Lexicon and Grammar, Houghton Mifflin, Boston.

Friedman, J. (1971). A Computer Model of Transformational Grammar, Elsevier.

Fu, K.S. (1982). Syntactic pattern recognition and applications, Prentice Hall.

Galil, Z.(1978) "Palindrome Recognition in Real Time by a Multitape Turing Machine", Journal of Computer and System Sciences 16, pp. 140-157.

Galil, Z. and Giancarlo, R. (1986), "Improved String Matching with k Mismatches", ACM Sigact News, Vol. 17, nr. 4, Spring 1986.

Galil, Z. and Seiferas, J. (1978), "A Linear-Time On-line Recognition Algorithm for 'Palstar' ", JACM, Vol. 25, Nr. 1, pp.102-111.

Galil, Z. and Seiferas, J. (1980), "Saving Space in Fast String-Matching", Siam J. Comput., Vol. 9, Nr. 2, May 1980, pp. 417-438.

Gallaire, H. (1969), "Recognition Time of Context-Free Languages by On-line Turing Machines", Information and Control 15, pp. 288-295.

Garey, M.R. and Johnson, D.S. (1979). Computers and Intractability, Freeman, San Francisco.

Garside, R. and Leech, G. (1982), "Grammatical Tagging of the LOB Corpus: General Survey". In: Johansson, ed. (1982), pp. 110-117.

Geens, D., Engels, L.K. and Martin, W. (1975), "Leuven Drama Corpus and frequency List". Leuven: PAL, Institute of Applied Linguistics, University of Leuven.

Geller, M.M. and Harrison, M.A. (1977), "Characteristic parsing, a framework for producing compact deterministic parsers", parts I and II, J. Comput. System Sci. Vol. 14, nr. 3, pp. 267-317 and 318-343.

Gonzalez, R. and Thomason, M.G. (1978). Syntactic pattern recognition, Addison Wesley.

Graham, S.L. and Harrison, M.A. (1976). "Parsing of general context-free languages", Advances in Computers 14, pp. 77-185.

Graham, S.L., Harrison, M.A. and Ruzzo, W.L. (1980). "An improved context-free recognizer", ACM Transactions on Programming Languages and Systems 2, pp. 415-462.

Greenbaum, S. (1984), "Corpus Analysis and Elicitation Tests". In: Aarts and Meijs, eds. (1984), pp. 193-201.

Greibach, S.A. (1973), "The hardest context-free language", SIAM J. on Comput. 2, pp. 304-310.

Griffiths, T. and Petrick, S. (1965), "On the relative efficiency of CF grammar recognition", CACM, Vol. 8, nr. 5, May 1965, pp. 289-300.

Griswold, R.E. and Griswold, M.T. (1983). The Icon Programming Language. Prentice Hall.

Griswold, R.E. and Hanson, D.R. (1980), "An Alternative to the Use of Patterns in String Processing", ACM Transactions on Programming Languages and Systems, Vol. 2, nr. 2, April 1980, pp 153-172.

Guibas, L.J. and Odlyzko, A.M. (1980), "A new proof of the linearity of the Boyer-Moore string searching algorithm", Siam J. Comput., Vol. 9, nr. 4, November 1980.

Haan, P. de (1984), "Problem oriented Tagging of English Corpus Data". In: Aarts and Meijs, eds. (1984), pp. 95-122.

Hamilton, C.D., Kimberley, R. and Smith, C.H. (eds.) (1985). Text retrieval, a directory of software. Gower, Aldershot.

Hardgrave, W.T. (1980), "Ambiguity in Processing Boolean Queries on TDMS Tree Structures: A Study of Four Different Philosophies", IEEE Transactions on Software Engineering, Vol. SE-6, nr. 4, July 1980, pp. 357-372.

Harris, Z. (1962). String Analysis of Sentence Structure, Mouton, The Hague.

Harrison, P.G. (1981), "Efficient Table-Driven Implementation of the Finite State Machine". The Journal of Systems and Software 2, 201-211.

Heilbrunner, S. (1979), "On the definition of ELR(k) and ELL(k) Grammars", Acta Informatica, Vol. 11, pp. 169-176.

Henderson, P. (1980). Functional Programming, Prentice Hall.

Hobbs, J.R. and Grishman, R. (1976), "The Automatic Transformational Analysis of English Sentences : An Implementation", Intern. J. Computer Math., Section A, Vol. 5, pp. 267-283.

Hoffmann, C.M., and O'Donnell, M.J., "Pattern Matching in Trees", JACM, Vol 29, nr. 1, Jan. '82, pp. 68-95.

Hofstadter, D.R. (1979). Gödel, Escher, Bach: an Eternal Golden Braid, Basic Books, New York.

Hofland, K. and Johansson, S. (1982). Word Frequencies in British and American English, The Norwegian Computing Centre for the Humanities, Bergen.

Hooper, J.B. (1976). An Introduction to Natural Generative Phonology, Academic Press, New York.

Hopcroft, J.E. and Ullman, J.D. (1969). Formal Languages and Their Relation to Automata, Addison Wesley.

Hopcroft, J.E. and Ullman, J.D. (1979). Introduction to Automata Theory, Languages and Computation. Addison Wesley.

Horning, J.J. (1974), "LR grammars and analysers", in: Bauer F.L. and Eickel, J., eds., "Compiler Construction, an Advanced Course", Lecture Notes in Computer Science 21, Springer, pp. 85-108.

Icame News, ed.: Johansson, The Norwegian Computing Centre for the Humanities, Bergen.

Janssen, O. (1984). Lemmatisierte Konkordanz zu den Schweitzer Minnesängern, Indices zur deutschen Literatur, Bd. 17. Tübingen, Niemeyer.

Johansson, S. (1982), "Lob Corpus Bibliography". In: Johansson, ed. (1982), appendix.

Johansson, S., ed. (1982). Computer Corpora in English Language Research. The Norwegian Computing Centre for the Humanities, Bergen.

Johansson, S., Leech G. and Goodluck, H. (1978). Manual of Information to Accompany the Lancaster/Oslo-Bergen Corpus of British English, for Use with Digital Computers, Department of English, University of Oslo.

Johansson, S. and Jahr, M.C. (1982), "Grammatical Tagging of the Lob Corpus. Predicting Word Class from Word Endings". In: Johansson, ed. (1982), pp. 118-146.

Jouannaud, J.-P. (ed.) (1985). Rewriting Techniques and Applications, Lecture Notes in Computer Science, Vol. 63, Springer Verlag.

Jourdan, M (1984), "Strongly non-circular attribute grammars and their recursive evaluation", ACM SIGPLAN, Vol. 19, nr. 6, June 1984, pp. 81-93.

Karp, R.M., Miller, R.E. and Rosenberg, A.L. (1972), "Rapid Identification of Repeated Patterns in Strings, Trees and Arrays", Proc. 4th Ann. ACM Symp. Th. of Comp., pp. 125-136.

Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context-free languages. Sci. Rep. AF CRL-65-785, Air Force Cambridge Research Laboratory, Bedford, Mass.

Katayama, T. (1984), "Translation of attribute grammars into procedures", ACM Trans. Program. Lang. Syst. 6, pp. 345-369.

Kemp, R. (1984). Fundamentals of the Average Case Analysis of Particular Algorithms, Wiley-Teubner Series in Computer Science, Stuttgart.

Ken-Chih Liu (1981), "On string pattern matching: a new model with a polynomial time algorithm", Siam J. Comp., Vol 10, nr.1, Feb. 1981, pp. 118-140.

King, M. (1981), "Design characteristics of a machine translation system", 7th. International Joint Conference on Artificial Intelligence, pp. 43-46.

Kjellmer, G. (1982), "Some Problems relating to the Study of Collocations in the Brown Corpus". in: Johansson, ed. (1982), pp. 25-33.

Kjellmer, G. (1984), "Some Thoughts on Collocational Distinctiveness". In: Aarts and Meijs, eds. (1984), pp. 163-171.

Klint, P. (1985). A Study in String Processing Languages. Lecture Notes in Computer Science 205, Springer-Verlag.

Knuth, D.E. (1965), "On the translation of languages from left to right", Information and Control, Vol. 8, pp. 607-639.

Knuth, D.E. (1968), "Semantics of Context-Free Languages". Mathematical Systems Theory 2, pp. 127-145.

Knuth, D.E., Morris, J.H. and Pratt, V.R. (1977), "Fast pattern matching in Strings", SIAM J. Comput., Vol. 6, Nr. 2 , 1977, pp. 323-350.

Koster, C.H.A. (1970), "Affix Grammars". In: Peck, J.E., ed., ALGOL68 Implementation, North-Holland, pp. 95-109.

Kristensen, B.B. and Madsen, O.L. (1981), "Methods for Computing LALR(k) Lookahead", ACM Transactions on Programming Languages and Systems, Vol. 3, nr. 1, pp. 60-82.

Kron, H. (1975). Tree templates and subtree transformational grammars, Ph.D. Dissertation, Univ. of California, Santa Cruz, Calif., 1975.

Kunst, A.E. and Blank, G.D. (1982), "Processing Morphology: Words and Clichés". In (Bailey, 1982).

Lalonde, W.R. (1979), "Constructing LR Parsers for Regular Right Part Grammars", Acta Informatica 11, 1979, pp. 177-193.

Lalonde, W.R. (1981), "The Construction of Stack-Controlling LR Parsers for Regular Right Part Grammars", ACM Transactions on Programming Languages and Systems, Vol.3, Nr.2, April 1981, pp. 168-206.

Lang, B. (1974). Deterministic Techniques for Efficient Non-deterministic Parsers, Technical report nr. 72, IRIA Laboria, Le Chesnay, France. Also presented at the Second Colloquium on Automata, Languages and Programming, Saarbrücken, 1974.

Leech, G., Garside, R. and Atwell, E.S. (1983a), "The Automatic Grammatical Tagging of the LOB Corpus". In: Newsletter of the International Computer Archive of Modern English (ICAME NEWS), 7, pp. 13-33.

Leech, G., Garside, R. and Atwell, E.S. (1983b), "Recent Developments in the Use of Computer corpora in English Language Research". In: Transactions of the philological Society, 23-40.

Levelt, W.J.M. (1973). Formele grammatica's in linguistiek en taalpsychologie. Van Loghum Slaterus, Deventer.

Lewerentz, C. and Nagl, M. (1984), "A Formal Specification Language for Software Systems Defined by Graph Grammars". In : U. Pape (ed.), Graphtheoretic Concepts in Computer Science, Proceedings WG, Linz, 1984, pp. 224-241.

Lewi, J., De Vlaminck, K., Huens, J. and Huybrechts, M. (1979). A Programming Methodology in Compiler Construction, 2 vols., North Holland.

Lipkie, D.E. (1979). A compiler design for multiple independent processor computer, Ph.D. dissertation, Univ. Washington, Seattle, 1979.

Ljung, M. (1974). A Frequency Dictionary of English Morphemes, Data Linguistica 9, University of Goeteborg, AWE/Gebers, Stockholm.

Longman (1978) Dictionary of Contemporary English, Harlow&London.

Lorho, B., ed. (1984). Methods and Tools for Compiler Construction, Cambridge University Press, New York, 1984.

Maas, D. and Maegaard, B. (1984). Syntax and Semantics of the Eurotra Formalism, Preliminary Version, October 1984. To be obtained from the European Committee.

Madsen, O.L. and Kristensen, B.B (1976), "LR-Parsing of Extended Context Free Grammars", Acta Informatica 7 , 1976, pp. 61-73.

Manacher, G. (1975), "A New Linear-Time 'on-line' Algorithm for Finding the Smallest Initial Palindrome of a String", JACM, Vol. 22, Nr. 3, 1975, pp. 346-351.

Marcus, M.P. (1980). Theory of Syntactic Recognition for Natural Language. Cambridge, MIT Press, Mass.

Marshall, I. (1982), "Choice of Grammatical Word-Class without Global Syntactic Analysis for Tagging Words in the LOB Corpus", Dept. of Computer Studies, Univ. of Lancaster.

Maurer, H. (ed.) (1979). Automata, Languages and Programming, 6th Colloquium, Graz, Vol. 71 in Lecture Notes in Computer Science, July 1979.

Meijer, H. (1986). Programmar: a Translator generator, Bloembergen Santee, Nijmegen. Doctoral diseertation, University of Nijmegen.

Meijs, W.J. (1982), "Exploring Brown with Query". In: Johansson, ed. (1982), pp. 34-48.

Meijs, W.J. (1984), "You can do so if you want to". In: Aarts and Meijs, eds. (1984), pp. 141-162.

Menzel, W. (1984), "A Grapheme-to-Phoneme Transformation for German", Computers and Artificial Intelligence, 3 (1984), Nr. 3, pp. 223-234.

Mickunas, M.D. and Modry, J.A. (1978), "Automatic error recovery for LR parsers", CACM, vol. 21, pp. 459-465, June 1978.

Mickunas, M.D. and Schell, R.M. (1978), "Parallel compilation in a multiprocessor environment", in Proc. ACM 78, pp. 241-246.

Nijholt, A. (1983). Deterministic Top-down and Bottom-up Parsing : Historical Notes and Bibliographies. Mathematisch Centrum, Amsterdam.

Ogden, W. (1968), "A helpful result for providing inherent ambiguity", Math. Systems Theory, 2, pp.191-194.

Oostdijk, N. (1984). "An Extended Affix Grammar for the English Noun Phrase". In: Aarts and Meijs, eds. (1984), pp. 95-122.

Overmars, M.H. and Van Leeuwen, J. (1979). Rapid subtree identification revisited, Tech. Rep. CS-79-3, Univ. of Urecht, Utrecht, Netherlands.

Pager, D. (1977), "A practical general method for constructing LR(k) parsers", Acta Informatica, Vol. 7, 249-268.

Partsch, H. and Steinbruggen, R. (1983), "Program Transformation Systems", ACM Computing Surveys, Vol. 15, nr. 3, sep. 1983, pp. 199-236.

Pavlidis, T. (1977). Structural Pattern Recognition, Springer Verlag.

Penello, T.J. and DeRemer, F., "A forward move algorithm for LR error recovery", in Conf. Rec., 5th Ann. ACM Symp. Principles of Programming Language, 1978, pp. 241-243.

Plenckers, L.J. (1984). "A pattern recognition System in the Study of the Cantigas de Santa Maria". In: Musical Grammars and Computer Analysis, Modena 1984.

Pohlmann, W. (1983), "LR Parsing for Affix Grammars", Acta Informatica, Vol. 20, pp. 283-300.

Purdom, P.W., and Brown, C.A. (1981), "Parsing Extended LR(k) Grammars", Acta Informatica, Vol. 15, pp. 115-127.

Quirk, R. (1984), "Recent work on Adverbial Realisation and Position". In: Aarts and Meijs, eds. (1984), pp. 185-192.

Remmen, F. (1985), "Hoe vriendelijk zijn vraagtalen in het gebruik ?", Informatie, jrg. 27, nr. 7/8, juli/aug 1985, pp. 666-673.

Renkema, J. (1981), "De taal van Den Haag", Staatsuitgeverij, Den Haag. Doctoral dissertation, Free University of Amsterdam.

Renouf, A. (1984), "Corpus Development at Birmingham University". In: Aarts and Meijs, eds. (1984), pp. 3-39.

Riesbeck, C.K. (1978), "An Expectation-Driven Production System for Natural Language Understanding". In (Waterman and Hayes-Roth, 1978).

Rosenkrantz, D.J. (1967), "Programmed Grammars: a New Device for Generating Formal Languages", Conf. Rec., 8th IEEE Ann. Symp. Switching Automata Theory, Austin, Texas.

Ruzzo, W.L. (1978), "General Context-Free Language Recognition". Ph.D. dissertation, U.C. Berkeley.

Ruzzo, W.L. (1979), "On the complexity of general context-free language parsing and recognition". In (Maurer, 1979), pp. 489-497.

Rytter, W. (1980), "A correct preprocessing algorithm for Boyer-Moore string-searching", Siam J. Comput., Vol. 9, Nr. 3, August 1980, pp. 509-512.

Rytter, W. (1987), "Parallel time O(log n) recognition of unambiguous context free languages", Information and Control 73, april, pp. 75-86.

Sager, N. (1981), Natural Language Information Processing, Addison-Wesley.

Salton, G., Fox, E.A. and Wu, H. (1983), "Extended Boolean information retrieval", CACM 26, Nov. 1983, Vol. 11, pp. 1022-1036.

Schane, S.A. (1973). Generative Phonology, Prentice-Hall.

Schell, R.M. (1979), "Methods for Constructing parallel compilers for use in a multiprocessor environment", Ph.D. dissertation, Univ. Illinois, Urbana.

Schimpf, K.M. and Gallier, J.H. (1985), "Tree Pushdown Automata", Journal of Computer and System Sciences 30, pp. 25-40.

Schmucker, K.J. (1984). Fuzzy sets, natural language computations, and risk analysis, Computer Science Press, Rockville.

Schneider, H.J. and Ehrig, H. (1976), "Grammars on Partial Graphs", Acta Informatica, Vol. 6, pp. 297-316.

Seiferas, J.I. and Galil, Z. (1977), "Real-time recognition of substring repetition and reversal", Math. Systems Theory, Vol. 11, pp. 111-146.

Simon, H-U. (1983), "Pattern matching in Trees and Nets", Acta Informatica, Vol. 20, pp. 227-248.

Skolnik, J. (1982). L-Trees, Technical report of the Computer Dept. Fac. of Arts, Univ. of Amsterdam.

Slocum, J. (1985), "A Survey of machine translation: its History, Current Status, and Future Prospects", computational linguistics, Jan-March, pp. 1-17.

Smit, G. de V. (1982), "A Comparison of Three String Matching Algorithms", Software-Practice and Experience, Vol. 12, pp. 57-66.

Snell, B. (1979). Translating and the Computer, North Holland.

Stenström, A-B. (1984), "Discourse Tags". In: Aarts and Meijs, eds. (1984), pp. 65-81.

Svartvik, J. (1982), "London-Lund Corpus Bibliography". In: Johansson, ed. (1982), appendix.

Svartvik, J. and Quirk, R., eds. (1980), "A Corpus of English Conversation". In: Lund Studies in English, Vol. 63, CWK GLeerup, Lund.

Svartvik, J. and Eeg-Olofsson, M. (1982), "Tagging the London-Lund Corpus of Spoken English". In: Johansson, ed. (1982), pp. 85-109.

Svartvik, J., Eeg-Olofsson, M., Forsheden, O., Orestrom, B. and Thavenius, C. (1982), "Survey of Spoken English: report on Research 1975-81". In: Lund Studies in English, Vol. 63, CWK GLeerup, Lund.

Takaoka, T. and Amamiya, M. (1975), "On the ambiguity Function of Context-Free Languages", Systems, Computers, Controls, Vol.6, Nr. 1.

Thompson, K. (1968), "Regular expression search algorithm", CACM, Vol. 11, nr. 6, pp. 419-422.

Tomita, M. (1986), "Efficient Parsing for Natural Language", Kluwer, Boston.

Turnbull, C.J.M. (1975), "Deterministic Left to Right Parsing", Technical report CSRG-48, Computer Systems Research Group, Univ. of Toronto.

Uit den Boogaart, P.C. ed. (1975). Woordfrequenties in geschreven en gesproken Nederlands, Scheltema en Holkema, Utrecht.

Ullman, J.D. (1982). Principles of Database Systems, Computer Science Press, Rockville.

Valiant, L. (1975), "General context free recognition in less than cubic time", J. Compu. Syst. Sci. 10, pp. 308-315.

Van der Steen, G.J., ed. (1981). De computer in de letteren, Computer Dept. Fac. of Arts, Univ. of Amsterdam.

Van der Steen, G.J. (1982), "A Treatment of Queries in Large Text corpora ". In: Johansson, ed. (1982), pp. 49-65.

Van der Steen, G.J. (1984), "On the unification of Matching, Parsing and Retrieving in Text corpora ", ICAME News, nr. 8, May 1984, pp. 41-46.

Van der Steen, G.J. (1985), "Syntactic pattern recognition as a Data-Base Tool". In: Allen, R.F. (ed.): Data Bases in the Humanities and Social Sciences, Paradigm Press, Florida, 20 pp.

Van der Steen, G.J., Houwink ten Cate, Ph. H.J. and De Roos, J. (1981). Coding conventions for the computational treatment of Hittite clay tablets, Computer Dept. Fac. of Arts, Univ. of Amsterdam.

Van Halteren, H. (1985). A Linguistic Database. Technical report, Dept. of Engels-Amerikaans, University of Nijmegen.

Vigna, P.D. and Ghezzi, C. (1978), "Context-Free Graph Grammars", Information and Control, Vol. 37, pp. 207-233.

Walters, D.A. (1970), "Deterministic Context-Sensitive Languages", Information and Control, Vol. 17, 1970, part I: pp 14-40, part II: pp 41-61.

Waite, W.M. and Goos, G. (1984). Compiler Construction, Springer Verlag.

Warshall, S. (1962), "A Theorem on Boolean matrices", JACM 9, pp. 11-12.

Wijsenbeek-Olthuis, T. (1987). Achter de poorten van Delft, Verloren, Hilversum. Doctoral dissertation University of Amsterdam.

Winograd, T. (1983). Language as a Cognitive Process, Volume I: Syntax, Addison-Wesley.

Woods, W.A. (1970), "Transition Network Grammars for Natural Language Analysis". CACM, Vol. 13, nr. 10, pp. 591-606.

11. Index

CWI TRACTS

1 D.H.J. Epema. *Surfaces with canonical hyperplane sections.* 1984.

2 J.J. Dijkstra. *Fake topological Hilbert spaces and characterizations of dimension in terms of negligibility.* 1984.

3 A.J. van der Schaft. *System theoretic descriptions of physical systems.* 1984.

4 J. Koene. *Minimal cost flow in processing networks, a primal approach.* 1984.

5 B. Hoogenboom. *Intertwining functions on compact Lie groups.* 1984.

6 A.P.W. Böhm. *Dataflow computation.* 1984.

7 A. Blokhuis. *Few-distance sets.* 1984.

8 M.H. van Hoorn. *Algorithms and approximations for queueing systems.* 1984.

9 C.P.J. Koymans. *Models of the lambda calculus.* 1984.

10 C.G. van der Laan, N.M. Temme. *Calculation of special functions: the gamma function, the exponential integrals and error-like functions.* 1984.

11 N.M. van Dijk. *Controlled Markov processes; time-discretization.* 1984.

12 W.H. Hundsdorfer. *The numerical solution of nonlinear stiff initial value problems: an analysis of one step methods.* 1985.

13 D. Grune. *On the design of ALEPH.* 1985.

14 J.G.F. Thiemann. *Analytic spaces and dynamic programming: a measure theoretic approach.* 1985.

15 F.J. van der Linden. *Euclidean rings with two infinite primes.* 1985.

16 R.J.P. Groothuizen. *Mixed elliptic-hyperbolic partial differential operators: a case-study in Fourier integral operators.* 1985.

17 H.M.M. ten Eikelder. *Symmetries for dynamical and Hamiltonian systems.* 1985.

18 A.D.M. Kester. *Some large deviation results in statistics.* 1985.

19 T.M.V. Janssen. *Foundations and applications of Montague grammar, part 1: Philosophy, framework, computer science.* 1986.

20 B.F. Schriever. *Order dependence.* 1986.

21 D.P. van der Vecht. *Inequalities for stopped Brownian motion.* 1986.

22 J.C.S.P. van der Woude. *Topological dynamix.* 1986.

23 A.F. Monna. *Methods, concepts and ideas in mathematics: aspects of an evolution.* 1986.

24 J.C.M. Baeten. *Filters and ultrafilters over definable subsets of admissible ordinals.* 1986.

25 A.W.J. Kolen. *Tree network and planar rectilinear location theory.* 1986.

26 A.H. Veen. *The misconstrued semicolon: Reconciling imperative languages and dataflow machines.* 1986.

27 A.J.M. van Engelen. *Homogeneous zero-dimensional absolute Borel sets.* 1986.

28 T.M.V. Janssen. *Foundations and applications of Montague grammar, part 2: Applications to natural language.* 1986.

29 H.L. Trentelman. *Almost invariant subspaces and high gain feedback.* 1986.

30 A.G. de Kok. *Production-inventory control models: approximations and algorithms.* 1987.

31 E.E.M. van Berkum. *Optimal paired comparison designs for factorial experiments.* 1987.

32 J.H.J. Einmahl. *Multivariate empirical processes.* 1987.

33 O.J. Vrieze. *Stochastic games with finite state and action spaces.* 1987.

34 P.H.M. Kersten. *Infinitesimal symmetries: a computational approach.* 1987.

35 M.L. Eaton. *Lectures on topics in probability inequalities.* 1987.

36 A.H.P. van der Burgh, R.M.M. Mattheij (eds.). *Proceedings of the first international conference on industrial and applied mathematics (ICIAM 87).* 1987.

37 L. Stougie. *Design and analysis of algorithms for stochastic integer programming.* 1987.

38 J.B.G. Frenk. *On Banach algebras, renewal measures and regenerative processes.* 1987.

39 H.J.M. Peters, O.J. Vrieze (eds.). *Surveys in game theory and related topics.* 1987.

40 J.L. Geluk, L. de Haan. *Regular variation, extensions and Tauberian theorems.* 1987.

41 Sape J. Mullender (ed.). *The Amoeba distributed operating system: Selected papers 1984-1987.* 1987.

42 P.R.J. Asveld, A. Nijholt (eds.). *Essays on concepts, formalisms, and tools.* 1987.

43 H.L. Bodlaender. *Distributed computing: structure and complexity.* 1987.

44 A.W. van der Vaart. *Statistical estimation in large parameter spaces.* 1988.

45 S.A. van de Geer. *Regression analysis and empirical processes.* 1988.

46 S.P. Spekreijse. *Multigrid solution of the steady Euler equations.* 1988.

47 J.B. Dijkstra. *Analysis of means in some non-standard situations.* 1988.

48 F.C. Drost. *Asymptotics for generalized chi-square goodness-of-fit tests.* 1988.

49 F.W. Wubs. *Numerical solution of the shallow-water equations.* 1988.

50 F. de Kerf. *Asymptotic analysis of a class of perturbed Korteweg-de Vries initial value problems.* 1988.

51 P.J.M. van Laarhoven. *Theoretical and computational aspects of simulated annealing.* 1988.

52 P.M. van Loon. *Continuous decoupling transformations for linear boundary value problems.* 1988.

53 K.C.P. Machielsen. *Numerical solution of optimal control problems with state constraints by sequential quadratic programming in function space.* 1988.

54 L.C.R.J. Willenborg. *Computational aspects of survey data processing.* 1988.

55 G.J. van der Steen. *A program generator for recognition, parsing and transduction with syntactic patterns.* 1988.

MATHEMATICAL CENTRE TRACTS

1 T. van der Walt. *Fixed and almost fixed points.* 1963.

2 A.R. Bloemena. *Sampling from a graph.* 1964.

3 G. de Leve. *Generalized Markovian decision processes, part I: model and method.* 1964.

4 G. de Leve. *Generalized Markovian decision processes, part II: probabilistic background.* 1964.

5 G. de Leve, H.C. Tijms, P.J. Weeda. *Generalized Markovian decision processes, applications.* 1970.

6 M.A. Maurice. *Compact ordered spaces.* 1964.

7 W.R. van Zwet. *Convex transformations of random variables.* 1964.

8 J.A. Zonneveld. *Automatic numerical integration.* 1964.

9 P.C. Baayen. *Universal morphisms.* 1964.

10 E.M. de Jager. *Applications of distributions in mathematical physics.* 1964.

11 A.B. Paalman-de Miranda. *Topological semigroups.* 1964.

12 J.A.Th.M. van Berckel, H. Brandt Corstius, R.J. Mokken, A. van Wijngaarden. *Formal properties of newspaper Dutch.* 1965.

13 H.A. Lauwerier. *Asymptotic expansions.* 1966, out of print; replaced by MCT 54.

14 H.A. Lauwerier. *Calculus of variations in mathematical physics.* 1966.

15 R. Doornbos. *Slippage tests.* 1966.

16 J.W. de Bakker. *Formal definition of programming languages with an application to the definition of ALGOL 60.* 1967.

17 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 1.* 1968.

18 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 2.* 1968.

19 J. van der Slot. *Some properties related to compactness.* 1968.

20 P.J. van der Houwen. *Finite difference methods for solving partial differential equations.* 1968.

21 E. Wattel. *The compactness operator in set theory and topology.* 1968.

22 T.J. Dekker. *ALGOL 60 procedures in numerical algebra, part 1.* 1968.

23 T.J. Dekker, W. Hoffmann. *ALGOL 60 procedures in numerical algebra, part 2.* 1968.

24 J.W. de Bakker. *Recursive procedures.* 1971.

25 E.R. Paërl. *Representations of the Lorentz group and projective geometry.* 1969.

26 European Meeting 1968. *Selected statistical papers, part I.* 1968.

27 European Meeting 1968. *Selected statistical papers, part II.* 1968.

28 J. Oosterhoff. *Combination of one-sided statistical tests.* 1969.

29 J. Verhoeff. *Error detecting decimal codes.* 1969.

30 H. Brandt Corstius. *Exercises in computational linguistics.* 1970.

31 W. Molenaar. *Approximations to the Poisson, binomial and hypergeometric distribution functions.* 1970.

32 L. de Haan. *On regular variation and its application to the weak convergence of sample extremes.* 1970.

33 F.W. Steutel. *Preservation of infinite divisibility under mixing and related topics.* 1970.

34 I. Juhász, A. Verbeek, N.S. Kroonenberg. *Cardinal functions in topology.* 1971.

35 M.H. van Emden. *An analysis of complexity.* 1971.

36 J. Grasman. *On the birth of boundary layers.* 1971.

37 J.W. de Bakker, G.A. Blaauw, A.J.W. Duijvestijn, E.W. Dijkstra, P.J. van der Houwen, G.A.M. Kamsteeg-Kemper, F.E.J. Kruseman Aretz, W.L. van der Poel, J.P. Schaap-Kruseman, M.V. Wilkes, G. Zoutendijk. *MC-25 Informatica Symposium.* 1971.

38 W.A. Verloren van Themaat. *Automatic analysis of Dutch compound words.* 1972.

39 H. Bavinck. *Jacobi series and approximation.* 1972.

40 H.C. Tijms. *Analysis of (s,S) inventory models.* 1972.

41 A. Verbeek. *Superextensions of topological spaces.* 1972.

42 W. Vervaat. *Success epochs in Bernoulli trials (with applications in number theory).* 1972.

43 F.H. Ruymgaart. *Asymptotic theory of rank tests for independence.* 1973.

44 H. Bart. *Meromorphic operator valued functions.* 1973.

45 A.A. Balkema. *Monotone transformations and limit laws.* 1973.

46 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 1: the language.* 1973.

47 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 2: the compiler.* 1973.

48 F.E.J. Kruseman Aretz, P.J.W. ten Hagen, H.L. Oudshoorn. *An ALGOL 60 compiler in ALGOL 60, text of the MC-compiler for the EL-X8.* 1973.

49 H. Kok. *Connected orderable spaces.* 1974.

50 A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker (eds.). *Revised report on the algorithmic language ALGOL 68.* 1976.

51 A. Hordijk. *Dynamic programming and Markov potential theory.* 1974.

52 P.C. Baayen (ed.). *Topological structures.* 1974.

53 M.J. Faber. *Metrizability in generalized ordered spaces.* 1974.

54 H.A. Lauwerier. *Asymptotic analysis, part 1.* 1974.

55 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 1: theory of designs, finite geometry and coding theory.* 1974.

56 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 2: graph theory, foundations, partitions and combinatorial geometry.* 1974.

57 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 3: combinatorial group theory.* 1974.

58 W. Albers. *Asymptotic expansions and the deficiency concept in statistics.* 1975.

59 J.L. Mijnheer. *Sample path properties of stable processes.* 1975.

60 F. Göbel. *Queueing models involving buffers.* 1975.

63 J.W. de Bakker (ed.). *Foundations of computer science.* 1975.

64 W.J. de Schipper. *Symmetric closed categories.* 1975.

65 J. de Vries. *Topological transformation groups, 1: a categorical approach.* 1975.

66 H.G.J. Pijls. *Logically convex algebras in spectral theory and eigenfunction expansions.* 1976.

68 P.P.N. de Groen. *Singularly perturbed differential operators of second order.* 1976.

69 J.K. Lenstra. *Sequencing by enumerative methods.* 1977.

70 W.P. de Roever, Jr. *Recursive program schemes: semantics and proof theory.* 1976.

71 J.A.E.E. van Nunen. *Contracting Markov decision processes.* 1976.

72 J.K.M. Jansen. *Simple periodic and non-periodic Lamé functions and their applications in the theory of conical waveguides.* 1977.

73 D.M.R. Leivant. *Absoluteness of intuitionistic logic.* 1979.

74 H.J.J. te Riele. *A theoretical and computational study of generalized aliquot sequences.* 1976.

75 A.E. Brouwer. *Treelike spaces and related connected topological spaces.* 1977.

76 M. Rem. *Associons and the closure statement.* 1976.

77 W.C.M. Kallenberg. *Asymptotic optimality of likelihood ratio tests in exponential families.* 1978.

78 E. de Jonge, A.C.M. van Rooij. *Introduction to Riesz spaces.* 1977.

79 M.C.A. van Zuijlen. *Emperical distributions and rank statistics.* 1977.

80 P.W. Hemker. *A numerical study of stiff two-point boundary problems.* 1977.

81 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 1.* 1976.

82 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 2.* 1976.

83 L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system.* 1979.

84 H.L.L. Busard. *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?), books vii-xii.* 1977.

85 J. van Mill. *Supercompactness and Wallman spaces.* 1977.

86 S.G. van der Meulen, M. Veldhorst. *Torrix I, a programming system for operations on vectors and matrices over arbitrary fields and of variable size.* 1978.

88 A. Schrijver. *Matroids and linking systems.* 1977.

89 J.W. de Roever. *Complex Fourier transformation and analytic functionals with unbounded carriers.* 1978.

90 L.P.J. Groenewegen. *Characterization of optimal strategies in dynamic games.* 1981.

91 J.M. Geysel. *Transcendence in fields of positive characteristic.* 1979.

92 P.J. Weeda. *Finite generalized Markov programming.* 1979.

93 H.C. Tijms, J. Wessels (eds.). *Markov decision theory.* 1977.

94 A. Bijlsma. *Simultaneous approximations in transcendental number theory.* 1978.

95 K.M. van Hee. *Bayesian control of Markov chains.* 1978.

96 P.M.B. Vitányi. *Lindenmayer systems: structure, languages, and growth functions.* 1980.

97 A. Federgruen. *Markovian control problems; functional equations and algorithms.* 1984.

98 R. Geel. *Singular perturbations of hyperbolic type.* 1978.

99 J.K. Lenstra, A.H.G. Rinnooy Kan, P. van Emde Boas (eds.). *Interfaces between computer science and operations research.* 1978.

100 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1.* 1979.

101 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2.* 1979.

102 D. van Dulst. *Reflexive and superreflexive Banach spaces.* 1978.

103 K. van Harn. *Classifying infinitely divisible distributions by functional equations.* 1978.

104 J.M. van Wouwe. *Go-spaces and generalizations of metrizability.* 1979.

105 R. Helmers. *Edgeworth expansions for linear combinations of order statistics.* 1982.

106 A. Schrijver (ed.). *Packing and covering in combinatorics.* 1979.

107 C. den Heijer. *The numerical solution of nonlinear operator equations by imbedding methods.* 1979.

108 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 1.* 1979.

109 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 2.* 1979.

110 J.C. van Vliet. *ALGOL 68 transput, part I: historical review and discussion of the implementation model.* 1979.

111 J.C. van Vliet. *ALGOL 68 transput, part II: an implementation model.* 1979.

112 H.C.P. Berbee. *Random walks with stationary increments and renewal theory.* 1979.

113 T.A.B. Snijders. *Asymptotic optimality theory for testing problems with restricted alternatives.* 1979.

114 A.J.E.M. Janssen. *Application of the Wigner distribution to harmonic analysis of generalized stochastic processes.* 1979.

115 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 1.* 1979.

116 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 2.* 1979.

117 P.J.M. Kallenberg. *Branching processes with continuous state space.* 1979.

118 P. Groeneboom. *Large deviations and asymptotic efficiencies.* 1980.

119 F.J. Peters. *Sparse matrices and substructures, with a novel implementation of finite element algorithms.* 1980.

120 W.P.M. de Ruyter. *On the asymptotic analysis of large-scale ocean circulation.* 1980.

121 W.H. Haemers. *Eigenvalue techniques in design and graph theory.* 1980.

122 J.C.P. Bus. *Numerical solution of systems of nonlinear equations.* 1980.

123 I. Yuhász. *Cardinal functions in topology - ten years later.* 1980.

124 R.D. Gill. *Censoring and stochastic integrals.* 1980.

125 R. Eising. *2-D systems, an algebraic approach.* 1980.

126 G. van der Hoek. *Reduction methods in nonlinear programming.* 1980.

127 J.W. Klop. *Combinatory reduction systems.* 1980.

128 A.J.J. Talman. *Variable dimension fixed point algorithms and triangulations.* 1980.

129 G. van der Laan. *Simplicial fixed point algorithms.* 1980.

130 P.J.W. ten Hagen, T. Hagen, P. Klint, H. Noot, H.J. Sint, A.H. Veen. *ILP: intermediate language for pictures.* 1980.

131 R.J.R. Back. *Correctness preserving program refinements: proof theory and applications.* 1980.

132 H.M. Mulder. *The interval function of a graph.* 1980.

133 C.A.J. Klaassen. *Statistical performance of location estimators.* 1981.

134 J.C. van Vliet, H. Wupper (eds.). *Proceedings international conference on ALGOL 68.* 1981.

135 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part I.* 1981.

136 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part II.* 1981.

137 J. Telgen. *Redundancy and linear programs.* 1981.

138 H.A. Lauwerier. *Mathematical models of epidemics.* 1981.

139 J. van der Wal. *Stochastic dynamic programming, successive approximations and nearly optimal strategies for Markov decision processes and Markov games.* 1981.

140 J.H. van Geldrop. *A mathematical theory of pure exchange economies without the no-critical-point hypothesis.* 1981.

141 G.E. Welters. *Abel-Jacobi isogenies for certain types of Fano threefolds.* 1981.

142 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 1.* 1981.

143 J.M. Schumacher. *Dynamic feedback in finite- and infinite-dimensional linear systems.* 1981.

144 P. Eijgenraam. *The solution of initial value problems using interval arithmetic; formulation and analysis of an algorithm.* 1981.

145 A.J. Brentjes. *Multi-dimensional continued fraction algorithms.* 1981.

146 C.V.M. van der Mee. *Semigroup and factorization methods in transport theory.* 1981.

147 H.H. Tigelaar. *Identification and informative sample size.* 1982.

148 L.C.M. Kallenberg. *Linear programming and finite Markovian control problems.* 1983.

149 C.B. Huijsmans, M.A. Kaashoek, W.A.J. Luxemburg, W.K. Vietsch (eds.). *From A to Z, proceedings of a symposium in honour of A.C. Zaanen.* 1982.

150 M. Veldhorst. *An analysis of sparse matrix storage schemes.* 1982.

151 R.J.M.M. Does. *Higher order asymptotics for simple linear rank statistics.* 1982.

152 G.F. van der Hoeven. *Projections of lawless sequences.* 1982.

153 J.P.C. Blanc. *Application of the theory of boundary value problems in the analysis of a queueing model with paired services.* 1982.

154 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part I.* 1982.

155 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part II.* 1982.

156 P.M.G. Apers. *Query processing and data allocation in distributed database systems.* 1983.

157 H.A.W.M. Kneppers. *The covariant classification of two-dimensional smooth commutative formal groups over an algebraically closed field of positive characteristic.* 1983.

158 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 1.* 1983.

159 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 2.* 1983.

160 A. Rezus. *Abstract AUTOMATH.* 1983.

161 G.F. Helminck. *Eisenstein series on the metaplectic group, an algebraic approach.* 1983.

162 J.J. Dik. *Tests for preference.* 1983.

163 H. Schippers. *Multiple grid methods for equations of the second kind with applications in fluid mechanics.* 1983.

164 F.A. van der Duyn Schouten. *Markov decision processes with continuous time parameter.* 1983.

165 P.C.T. van der Hoeven. *On point processes.* 1983.

166 H.B.M. Jonkers. *Abstraction, specification and implementation techniques, with an application to garbage collection.* 1983.

167 W.H.M. Zijm. *Nonnegative matrices in dynamic programming.* 1983.

168 J.H. Evertse. *Upper bounds for the numbers of solutions of diophantine equations.* 1983.

169 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 2.* 1983.