Bijgevoegde stelling bij

Daelemans, W. (1987) *Studies in Language Technology: An Object-Oriented Computer Model of Morphophonological Aspects of Dutch.*

Excessive copying of descriptions in parsing with unification-based grammar formalisms can be prevented by using a system based on the propagation of cancellations in a self-updating dependency network.

WD.

KATHOLIEKE UNIVERSITEIT LEUVEN
DEPARTEMENT LINGUISTIEK

STUDIES IN LANGUAGE TECHNOLOGY

An Object-Oriented Computer Model of Morphophonological Aspects of Dutch

WALTER DAELEMANS

Proefschrift aangeboden ter verkrijging
van de graad van doctor in de Letteren
en de Wijsbegeerte
Promotor: Prof. Dr. F. G. Droste
Co-promotor: Prof. Dr. G. A. M. Kempen

Leuven, 3 april 1987

SAMENVATTING

Het menselijke taalgedrag kan worden opgevat als een op kennis gebaseerde probleemoplossende activiteit. Wanneer een mens de relatie legt tussen betekenis en klank en vice versa, dan voert hij een aantal, meestal onbewuste, redeneerprocessen op een aantal kennisbronnen uit. We kunnen deze vaardigheid simuleren (of imiteren) door computermodellen te bouwen waarbij de nodige kennis door datastructuren wordt gerepresenteerd, en processen door programma's die van deze datastructuren gebruik maken. Voordelen van deze aanpak zijn aan de ene kant consistentie en volledigheid (voor de theoretische taalkunde), en aan de andere kant nuttige applicaties (voor de toegepaste taalkunde). In deze dissertatie proberen we enkele aspecten van het menselijke taalgedrag op deze *computationele* manier te benaderen.

We gaan uit van een kort overzicht van verschillende disciplines die op een of andere manier een relatie leggen tussen de studie van de taal en de computerwetenschap. We richten ons daarbij vooral op de doelstellingen en de methodologie van de *taaltechnologie*, het deel van de computerlinguistiek dat zich bezig houdt met toepassingen. We proberen aan te tonen dat het paradigma van het objectgericht programmeren uitstekend geschikt is om linguistische kennis en processen te representeren. Alle programmeerparadigma's zijn equivalent omdat de programma's die zij genereren uiteindelijk allemaal Turing-machine berekenbaar zijn, maar voor de programmeur (en dus ook voor de programmerende taalkundige) zijn ze verschillend omdat ze verschillende metaforen suggereren om het probleemdomein te conceptualiseren. In een objectgerichte programmeerstijl worden alle concepten, entiteiten en gebeurtenissen in een domein als computationele objecten voorgesteld. Alle kennis, zowel declaratief als procedureel wordt opgeslagen in het object waar ze betrekking op heeft, en is uitsluitend via dat object bereikbaar. We geven een aantal computationele en linguistische argumenten ten voordele van objectgericht programmeren, en stellen een geavanceerd objectgericht kennisrepresentatiesysteem voor.

We passen de objectgerichte filosofie toe op enkele aspecten van de Nederlandse fonologie en morfologie. We hebben onze aandacht beperkt tot de synthese van werkwoordsvormen, de analyse van samenstellingen, de detectie van interne woordgrenzen en lettergreepgrenzen, en fonematiseringsalgoritmen. De nadruk in deze beschrijving ligt vooral op de interactie tussen morfologische, fonologische en lexicale representaties en op de mogelijkheid tot uitbreiding van de ontwikkelde kennisbank. We geven ook een aantal beschouwingen weer over het ontwerp van een lexicale databank voor taaltechnologische toepassingen.

De resulterende morfo-fonologische kennisbank kan op veel manieren gebruikt worden in toepassingen. We bespreken het concept van een *auteursomgeving* waarmee we een verzameling interagerende programma's bedoelen die het leven van de gebruiker van tekstverwerkers aangenamer maken. Twee van de modules in zo'n auteursomgeving: automatische woordafbreking en automatische detectie en correctie van spel- en typefouten worden in detail behandeld. We stellen programma's voor die een oplossing bieden voor de problemen die voortkomen uit de manier waarop in het Nederlands samenstellingen worden gevormd. Wanneer onvolledigheden in de kennisbank een volledige oplossing voor sommige sub-problemen onmogelijk maken suggereren we heuristieken. Heuristieken worden trouwens ook gebruikt om de efficiëntie van de ontwikkelde programma's te verhogen.

Een domein in de Kunstmatige Intelligentie dat vlug aan belang wint is het *intelligent computergesteund onderwijs*. Een intelligent systeem voor computergesteund onderwijs bevat naast kennis over de leerstof die moet worden onderwezen ook een model van de leerling, heuristieken voor de diagnose van de fouten van de leerling, een module die gemaakte fouten uitlegt, en educatieve strategieën. We hebben een prototype van zo een systeem gebouwd voor het aanleren van een bepaald aspect van de Nederlandse spelling (de spelling van de werkwoordsvormen).

Systemen om regels te testen bieden een fundamenteel nieuwe manier om taalkunde te bedrijven. Ze versnellen de ontwikkeling van regelsystemen en theorieën en voorzien de taalkundige van krachtige methodes om complexe interacties en neveneffecten van regels te controleren. We beschrijven het prototype van een dergelijk systeem voor het testen van fonologische regels. We geven eveneens een voorbeeld van hoe de linguistische algoritmen die we hebben ontwikkeld toegepast kunnen worden in de lexicografie. We schetsen een experimentele omgeving waarin de lexicograaf op een gemakkelijke manier lexicale databanken kan creëren, uitbreiden en veranderen. We

schenken ook aandacht aan de manieren waarop ons morfo-fonologisch model zou kunnen worden gebruikt als module in meer uitgebreide systemen. Een morfologische component is onontbeerlijk in systemen voor automatische vertaling en in dialoogsystemen als deel van de syntactische analyse- en syntheseprogramma's. Een fonologische module is essentieel in elk systeem dat taal wil verwerken met spraaksignalen als input of output. De transportabiliteit en de modulariteit van objectgericht geprogrammeerde systemen maakt hen uiterst geschikt voor integratie in grotere systemen. We bespreken meer bepaald de mogelijke rol van ons fonematiseringsalgoritme in een spraaksynthesesysteem.

ABSTRACT

This dissertation presents a computer model of aspects of Dutch morphology and phonology. After a concise introduction to language technology as a part of Artificial Intelligence, it is argued that the object-oriented programming paradigm is ideally suited to represent linguistic knowledge and processes. An object-oriented implementation of aspects of Dutch morphology (word form synthesis and recognition) and phonology (syllabification, phonemisation, phonological rules) is presented to support this opinion. It is shown how this morphophonological module can be used to provide a principled solution to some problems in word level language technology (notably automatic hyphenation and spelling/typing error correction) for which only a defective solution can be given using traditional (engineering) approaches. The utility of the module in the development of other applications is discussed. Among those, prototypes of the following were implemented: an Intelligent Tutoring System for some aspects of Dutch spelling, an environment for the creation and testing of complex systems of linguistic rules and a lexicographic tool for the creation, updating and extending of lexical databases.

TABLE OF CONTENTS

Chapter 8 Miscellaneous Applications

PREFACE

Language technology can be situated at the intersection of linguistics, psychology and computer science. It concerns itself with the development of computer programs which produce, understand and manipulate natural language. As a technology, it should produce artificial translators, artificial dialogue partners, artificial editors and artificial language teachers. As a science, it should provide an instrument to construct and evaluate linguistic and psycholinguistic theories about language structure and language use. Figure 1 shows the different modules which play a role in a natural language processing system.

Input to a language processing system can be either spoken or written text. In the first case, acoustic signals must be transformed into a computer-readable representation. This process needs the concerted action of several linguistic components (putting it in a single box as in the figure is a simplification). Input text is analysed at the word-level by a word parsing algorithm computing the internal structure of words, and at the sentence-level by a sentence parsing algorithm computing the syntactic structure of sentences. The semantic representation of an input text is computed using the syntactic and morphological representations, the lexical meaning of individual morphemes and additional information from domain knowledge and contextual knowledge. Advanced systems would also include a level of discourse analysis. In generating language from a semantic representation, syntactic and morphological generation modules are used to produce written text. Additional intonation, phonemisation (grapheme-to-phoneme transliteration), syllabification (computing syllable boundaries), and phonological rule components (summarised in a single box in the figure) are necessary to compute a phonetic representation detailed enough to be used by a speech synthesiser to produce natural-sounding speech. All analysis and generation components make extensive use of the lexical database containing the inventory of the morphemes of a language and their associated phonological, morphological, syntactic and semantic information.

In this dissertation, I will be concerned only with those modules which are shaded in Figure 1: the linguistic levels at and beneath the word level. The text is divided into three parts and eight chapters. Part I is devoted to methodological

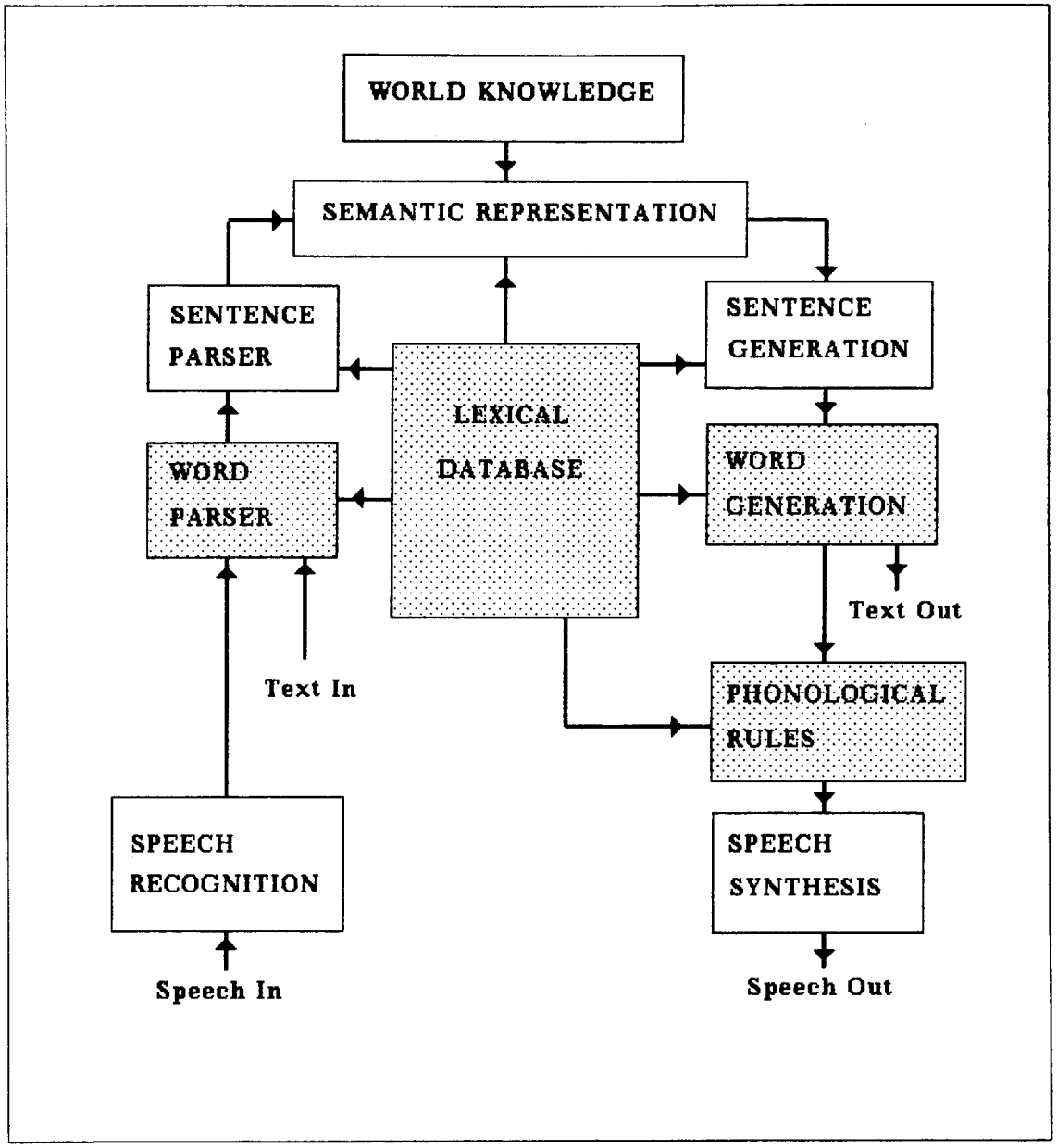


Figure 1. Modules in a language processing system.

considerations. In Chapter 1, a concise introduction to the field of language technology is given. It is argued that language technology should adopt the methodological principles of computational linguistics proper, and not content itself with pure engineering, as is often the case. Solutions to problems should be theoretically motivated. This point of view was not dictated by a love of theoretical purity, but by a cool observation of the failure of an engineering approach. Chapter 2 describes the object-oriented programming paradigm and its benefits for programming in general and for programming linguistic knowledge and processes in particular. The

central claim in this chapter is that, although all programming paradigms are ultimately equal in terms of Turing computability, they are very different in their adequacy as notations for describing information and action in a particular domain. Part II is a description of our morphophonological module for Dutch. Different knowledge sources and processes in this domain are described in an object-oriented way. Algorithms to compute internal morphological boundaries, inflected forms of verbs, syllable boundaries, and a phoneme representation of spelling are described in Chapters 3 and 4. Part III describes a number of applications of the model in Part II: Automatic hyphenation, automatic spelling and typing error detection and correction, intelligent computer-assisted instruction, lexicographic tools, text-to-speech systems, linguistic rule-testing devices etc.

The sequence in which the chapters of this dissertation are ordered is slightly misleading. It may suggest to the reader that I started from some methodological premises, selected a domain, constructed a computational model to account for some phenomena in this domain, and finally developed applications to investigate the usefulness of the model. In fact, the reported research started from two practical problems in the framework of word processing: automatic spelling error detection and automatic hyphenation for Dutch. It soon became clear that the solutions provided in the literature were adequate for English, but not for Dutch, due to the peculiar compounding behaviour of Dutch morphemes. This led to the insight that a principled solution to the hyphenation and detection problem would involve a level of morphological analysis. However, the slowness of existing morphological analysis programs made them useless in practical word processing applications. One solution was to place part of the burden of analysis on the lexical database, which should contain wordforms, and not only morphemes. We could suffice then with a fast wordform parser to analyse compounds, provided we developed a morphological synthesis program to construct and update the wordform dictionary automatically. From these considerations, the usefulness of a modular, portable and complete model of at least parts of Dutch morphology became obvious. Interaction between morphology and phonology introduced new problems and new requirements for the model, for instance the fact that it should include a phonological level as well. After implementing this level, more applications (such as grapheme-to-phoneme transliteration) became feasible, and so on. A lot of work remains to be done to obtain a complete model of all aspects of Dutch morphology and phonology, but we believe that the approach taken here, and the programs developed, make completion of such a model a matter of time rather than of invention.

Implementation Note. Development of the programs began early 1984. Various versions exist, written in different languages and dialects (Franz Lisp, ZetaLisp, NIL Lisp, Orbit, Flavors and KRS) and running on different machines (Vax, Sun and Symbolics Lisp Machine). I am presently working on the integration of all relevant software into a single package written in Common Lisp and KRS and running on a Symbolics Lisp Machine. Source code listings are available for research purposes upon request.

Production Note. The text was produced using the UNIX Troff formatting system. The more elaborated pictures were drawn using the Symbolics Lisp Machine picture editor.

Acknowledgements. This work was financially supported by the EC under ESPRIT project OS-82 in which I participated both in Nijmegen and Brussels, and by a research grant from the Flemish Directorate for International Cultural Co-operation.

First, I would like to thank my thesis supervisors, Prof. F.G. Droste and Prof. G. Kempen for giving me the opportunity to write this dissertation, and for their inspiring comments on my work. At the Katholieke Universiteit Leuven, Prof. Droste awoke a deep interest in me for theoretical linguistics. His clear view on the subject and his enthusiasm have always been an important incentive for me to go on. Prof. Franz Loosen of the psychology department of the Katholieke Universiteit Leuven broadened my horizon to psycholinguistics and artificial intelligence, an evolution which I have never regretted. His encouragement has meant a lot to me. He also introduced me to Prof. Gerard Kempen of the psychological laboratory of the Katholieke Universiteit Nijmegen, under whose supervision I started working on the computer programs which finally led to this dissertation. I owe a lot to his incessant willingness to help me and to share his unparalleled knowledge of language technology. At the Artificial Intelligence Laboratory of the Vrije Universiteit Brussel, I finished this dissertation. I am very grateful to Prof. Luc Steels for allowing me to complete my work in peace in an intellectually stimulating environment.

I am pleased to acknowledge the detailed, precise and useful critiques on the text which I received from the following people: Flip Droste, Gerard Kempen, Luc Steels, Carel Van Wijk, Koen De Smedt, Ronnie Geluykens, Viviane Jonckers and Marina Geerts. I have also profited from discussions on the subject with former and present colleagues, especially Henk Schotel, Dik Bakker and Bernard Manderick. Thanks are due to Eric 'Wyb' Wybouw for sharing his wizzardly knowledge about

Lisp, Unix and Troff. I am grateful to my parents for their past financial and their continuous moral support. Finally, I would like to thank my wife, Marina, for her patience, her encouragement and her love. This dissertation is dedicated to her.

PART I

METHODOLOGY

Human verbal behaviour can be viewed as a knowledge-based problem-solving activity. In mapping sound to meaning and vice versa, a person applies (unconscious) reasoning processes to a variety of knowledge sources (collections of highly structured information). This capacity can be simulated (or imitated) by constructing computer models in which knowledge is represented by data structures, and processes by programs driven by these data. Advantages of this approach are consistency and completeness of description for theoretical linguistics, and useful applications for applied linguistics. Different disciplines relating language studies and computer science will be reviewed in Chapter 1, and the goals and methodology of language technology in particular will be studied.

In Chapter 2, it will be argued that the object-oriented programming paradigm is ideally suited to represent linguistic knowledge and processes. All programming paradigms may be equivalent in that the programs they generate are ultimately Turing-machine equivalent, but to the programmer they are different in the metaphors they provide to conceptualise the problem domain. When using an object-oriented programming style, concepts, entities and events in the problem domain are represented as computational objects. All knowledge, procedural as well as declarative, is stored in and accessible through the object. The computer appears to the programmer as a set of autonomous processors which can be addressed independently. A number of computational and linguistic arguments in favour of object-oriented programming will be provided, and an advanced object-oriented knowledge representation system will be described.

CHAPTER 1

Language Technology

Several disciplines attempt to establish a relationship between natural language study and computer science. This general effort has become known under the name *natural language processing*. We will not be concerned here with the classification and counting of large quantities of linguistic data (*statistical* or *quantitative linguistics*), nor with the theory of formal languages and automata (*mathematical* or *algebraic* linguistics). The remaining disciplines are captured under the name *computational linguistics*. Computational linguistics, especially when dealing with the development of practical applications, is called *language technology*.

In the following section, we will describe the Artificial Intelligence approach to computational linguistics (section 1.1). Characteristic of this approach is that it interprets the study of language as part of *cognitive science*. Viewed from this perspective, computational linguistics includes *computational psycholinguistics* (the testing and implementation of psychological models of language processing), which we consequently do not grant the status of an autonomous discipline. In section 1.2, it will be argued that language technology needs no separate methodology in which the methodological constraints of *theoretical* computational linguistics are relaxed. This separate methodology is present explicitly or implicitly, however, in a lot of work in language technology.

1.1 The Artificial Intelligence Approach

> *The first twenty obvious ideas about how intelligence might work are too simple or wrong.*
> David Marr

In our opinion, the most fruitful approach to natural language processing is the one adopted in *Artificial Intelligence* research. AI can be defined as the science and technology of knowledge (see e.g. Steels, 1985). Knowledge is defined as information representing collections of highly structured objects (Kempen, 1983). More detailed definitions of AI can be found in any textbook on the subject (e.g. Winston, 1984; Charniak and McDermott, 1985).

In an AI-perspective, language is viewed as a *knowledge-based process* (e.g. Winograd, 1983); a cognitive ability which allows people to apply (unconscious) reasoning processes to stored linguistic, world and situational knowledge. This cognitive system is described in a *computer model* in which knowledge is represented by data structures, and mental processes are represented by programs using or manipulating these data structures. As regards knowledge representation and manipulation, no *a priori* distinction is made between the linguistic and other cognitive systems. This position is not necessarily conflicting with some form of the autonomy or modularity thesis (e.g. Fodor, 1983) which views language as a computational cognitive module exhibiting only constrained information exchanges with other modules. We can envision an autonomous module making use of general knowledge representation, problem solving and learning techniques, yet at the same time having its own structure and interacting in restricted ways with other modules.

The AI-approach makes use of a predominant axiom (or metaphor) from cognitive science: *the mind as an information processing system*. This metaphor states that human and machine intelligence can be described at an appropriate level of abstraction as the result of symbol-manipulating processes; i.e. both people and (intelligent) machines are instances of the same *informavore* species (Miller, 1984). Cognitive science is a multidisciplinary science bringing together psychology, philosophy, linguistics, neuroscience, educational science and artificial intelligence in an effort to build programs or programmable theories which simulate or describe a cognitive behaviour, taking into account empirical phenomena (see e.g. Adriaens, 1986 for a linguistic theory developed within this framework).

The basic relationships between cognitive system, computer model, algorithm and computational theory are sketched in Figure 1. The levels and interrelations were inspired by Marr (1977, 1982) and Kempen (1983).

A *computer model* is a program running on some machine and exhibiting the same behaviour as the human cognitive system described by it. If this is the case,

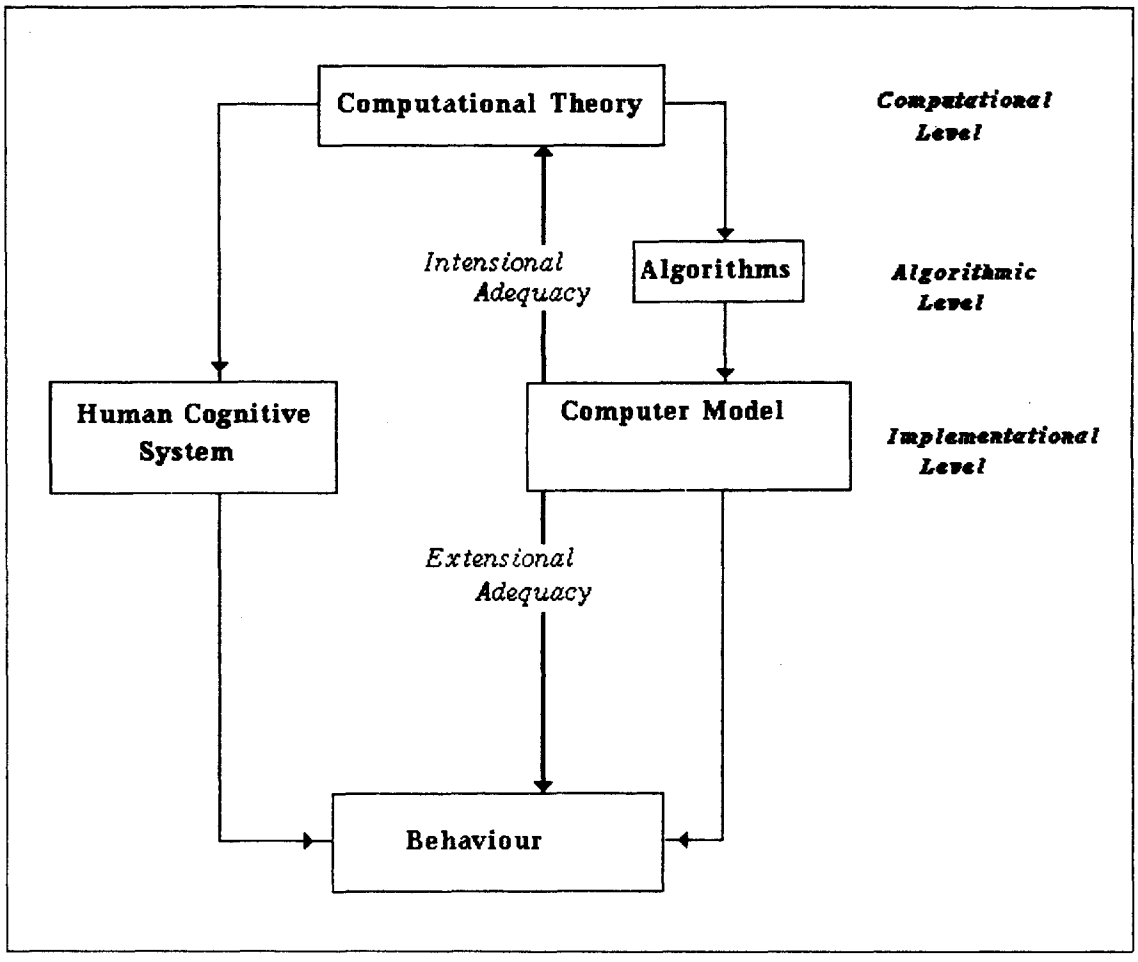


Figure 1: Relation between theory, algorithm, computer model and cognitive system.

we say the model is *extensionally* adequate. Computer model and biological system together form the *implementational level*; the level in which an abstract cognitive system is physically realised. The concept of a *model* always implies a hypothetical isomorphic (or homomorphic) relation between simulated system and simulating system. It is theoretically possible to posit this relation at the level of hardware: between central nervous system and computer architecture, respectively. Recent efforts at connectionist models of cognition (e.g. Ballard, 1986) may be an example of this, although some workers in this field locate their research at the algorithmic level which we will outline shortly (e.g. Rumelhart and McClelland, 1986). But most often, the isomorphic relation is postulated at the level of *computational theory*.

At that level we are concerned with an abstract and formal analysis of the problem (e.g. transforming sound into meaning in the case of language), and with the

computational functions necessary to solve it. According to Marr, Chomsky's *competence theory* is a computational theory in this sense. The computational level constitutes a *tertium comparationis* between human cognitive system and computer model. The computer model is *intensionally* adequate if it implements an internally coherent and complete theory at the computational level.

The *algorithmic level* is concerned with how the computational theory can be implemented. At this level, representations and processes transforming representations are basic building blocks. Efficiency considerations (in both processing time and storage requirements) and psychological data are especially relevant here. Performance theories are usually algorithmic in nature.

According to Marr (1977), AI-research should proceed in a three-step top-down manner. First, an interesting information processing *problem* is isolated. Next, a *computational theory* is developed describing and explaining the properties of the problem, and its basis in the physical world. Finally, *algorithms* are devised which implement a solution to the information processing problem in a computer model. This approach was applied by his group to low-level vision. Nevertheless, he also noted that there may be information processing problems for which no theory of computation could (yet) be developed. Hence his methodological preference for low-level vision problems like *stereo-vision*, as opposed to e.g. *object-recognition*. He suggested that most work in the field of AI has tended to be concerned with performance and with the construction of algorithms rather than with the development of computational theories[1]. In his view, however, information processing problems are more important than implementation mechanisms, and the theory of computation should precede the construction of algorithms.

We do not share this view. Language processing may be a problem for which no computational theory exists yet, as it lacks a clear basis in physiology and psychophysics. But this need not keep us from trying to construct one. We can use the evaluation measures[2] of linguistics and the experimental data of psychology to

[1] Marr (1977) gives a 'sociological' reason for this: external pressures for early results made researchers jump from the problem to an algorithm implementing the solution without an intermediate stage of theorising. Chomsky (1983) has added a 'psychoanalytic' interpretation: many researchers do not want to get into areas where there may be far-reaching and abstract principles (computational theory of intelligence) because they would like to believe that explanations are close to the surface.

[2] Evaluation measures are used to make a choice between different linguistic theories. They are based on such (vague) criteria as simplicity, adequacy, significance, learnability etc.

constrain our computer model. We can even use computational criteria like efficiency, speed, resource usage, etc., to restrict the model. This means that we see the algorithmic level as an equally useful inspiration to computer model building as the computational theory level. We will argue in Chapter 2 that some notations and formalisms (algorithmic level entities) are better suited to represent concepts in a problem domain than others, and that they may even have a definite influence on the characteristics of the theory developed. We will therefore regard the computational and the algorithmic levels as one level (the computational level) in the remainder of this dissertation.

At this (generalised) computational level, it is possible to theorise about language processing in the abstract (cp. Thompson, 1983, *theory of linguistic computation*). Moreover, design restrictions such as *modularity* can be postulated at this level. Modularity is a concept which is interpreted somewhat differently in various scientific disciplines. In *computer science*, a computer program is called modular if chunks of computer program can be added, modified or deleted, independently of the remainder of the database. A modular system or program is easier to understand and to maintain. In *linguistics*, modular theories are used to dissect a complex process into a system of simpler processes (e.g. the modularity of formal grammar in recent accounts of transformational generative grammar; Chomsky 1981). A complex system can be better understood as a set of modules interacting in clearly defined ways. Modularity has also proved to be a *psychologically* relevant property of low-level vision (e.g. Marr, 1982), and has been claimed for other cognitive systems as well (we have already mentioned Fodor, 1983). Thus, we can distinguish two aspects of modularity in computational theory: as a methodological principle to gain insight (even at the loss of efficiency or plausibility), and as a design property of biological cognitive systems. This constitutes a double incentive to develop computational theories and models which are modular.

The lack of a physical basis for the devising of representations and processes implies that we cannot be sure that the representations hypothesised are 'real' (i.e. that human beings possess and use them). But at least we have the advantage of extensional and intensional validity.

(Chomsky, 1965; Botha, 1978).

Notice that the relation between psychology and linguistics on the one hand, and the computational model on the other hand is not one of uni-directional influence. Just as psychological (e.g. Johnson-Laird, 1980) and linguistic theories provide a source of inspiration for the development of a computer model does the latter have a beneficial effect on the former as well. A computer model can bring more insight into the representations and processes used by people when communicating. It has even been claimed that much psychological research has been guided by AI research rather than vice versa (e.g. Winograd, 1980). Computer models can also be used as a tool in linguistic research. In section 1.2.2 we will return to the advantages for linguistics, Finally, the preoccupation with knowledge representation and processing in AI has led to new programming languages and computing paradigms (Prolog, object-oriented programming) in mainstream computer science.

Computational linguistics in the Dutch language area is divided between two unreconcilable interpretations of the discipline. The cognitive artificial intelligence approach has been adopted by a.o. Hoenkamp (1983, 1985), Kempen (1983), and Adriaens (1986). Other workers in the field of computational linguistics interpret natural language processing as the construction of algorithmic models of linguistic grammars and theories. This means that they see computational linguistics as a tool in the development of descriptive theories of language (a.o. Van Bakel, 1983; Brandt Corstius, 1974) and not as an effort to construct cognitive theories of verbal behaviour. Opposition to the AI-approach is often quite strong (e.g. Brandt Corstius, 1981).

1.2 Applications

> *The linguist should be grateful that even if he is not interested in 'practical' results, the requirements, say, of computer programming may be a healthy incentive for explicit and rigorous formulation.*
> G.C. Lepschy.

Language technology (linguistic engineering) can be defined as the part of computational linguistics which concerns itself with the development of (commercial) applications. Examples are advanced word processors, natural language interfaces to databases, automatic translation systems, etc.

The linguistic and psychological relevance of these applications is often small, as they are mostly developed by programmers whose main concern is short-term computational efficiency. They try to combine the highest possible accuracy with the highest possible processing speed and the smallest possible storage prerequisites. Their systems contain *ad hoc* rules and feature a predominant use of heuristics[3] instead of algorithms. This need not be a bad thing, since commercial applications should be efficient rather than theoretically adequate. It could be argued that in language technology, intensional validity or an isomorphic relation with human information processing are no prerequisites for the model as long as it is extensionally valid (i.e. if the program exhibits the desired behaviour).

However, we believe that this engineering approach often leads to systems which cannot properly do what they are supposed to do. The 'highest possible accuracy' (e.g. 75% sentences correctly translated, 95% correct hyphenations) may often be increased even further if the desired behaviour is viewed from the perspective of a computational theory (cp. Droste, 1969). One of the central claims in this dissertation is that algorithms and heuristics used in language technology should *also* be based on a computational theory which is founded in linguistics and psychology.[4] An important shortcoming of most technological approaches to language processing is their complete lack of theoretical foundation, resulting in inaccurate systems.

We will exemplify the unfruitfulness of the latter approach in Chapter 5, where we will show the inadequacy of a heuristic approach to hyphenation (Boot, 1984), and suggest an alternative approach, based on a computational theory of syllabification outlined in Chapter 4.

Boot adheres to what he calls a (result-oriented) *expert system* approach to computational linguistics. He interprets expert systems as systems of heuristic rules solving some specific problem, and claims that it is not necessary to have complete knowledge about human language competence in order to build natural language

[3] Heuristics are implemented as algorithms as well. However, the meaning we assign here to heuristics is the one traditionally used in AI: heuristics are informal judgemental rules drawing on regularity and continuity in the domain modeled (cf. Lenat, 1982). The knowledge they embody is therefore incomplete. An algorithm on the other hand embodies complete knowledge about how a problem is to be solved in principle. The algorithm may be deficient, but that is an entirely different matter.

[4] The call for a *theory of translation* in machine translation research (E.g. Van Eynde, 1985) may be a manifestation of the same concern.

processing systems. In our radically different view, a computational model, based on a computational theory and easily adaptable to different applications, is theoretically as well as practically superior to an approach in which each application has its own, independent, *ad hoc* 'expert system'[5]. Language technology in our opinion is the construction of computational models incorporating computational theories (as in AI), and the study of how these can be efficiently exploited in applications. Apart from the fact that applications are useful in their own right, they can also function as a test case for computational theories, suggesting extensions or modifications.

1.2.1 An Overview of Application Types

Five main branches of language technological applications may be distinguished:

(i) The computer as a *dialogue partner.* Natural Language front ends (accepting spoken or written language) make data processing systems more user-friendly and easy to use. The slogan here is *let the computer speak our language instead of vice versa.* Natural language interfaces have been or are being developed to databases (question answering systems) and expert systems (explanation modules). A 'talking computer' fits more naturally into people's lives and lowers the threshold to computer use. This is demonstrated in Figure 2 (adapted from Zoeppritz, 1983), which contrasts a database request in SQL (a database query language) with its natural language counterpart.

```
select all x member
   from emp x
     where x.member not in
       (select unique y.member
           from emp
              where y.city='antwerp')

Which members do not live in Antwerp?
```

Figure 2. A database request in SQL as opposed to natural language.

(ii) The computer as a *translator.* After a rude awakening from the dream of unrestricted full-automatic translation in the late sixties, more reasonable efforts at restricted computer-aided translation are being conducted, especially in Japan (Fifth Generation Computers program), and Europe (EC-Eurotra, Siemens-

[5] The structure of the expert system may not be ad hoc, but the rules used are.

Metal, BSO, Philips-Rosetta ...).

(iii) The computer as a *teacher*. An effective system for Computer Assisted Instruction will have to contain some kind of natural language interface to respond sensibly to the input from the user (cp. i). Furthermore, in CAI systems for the subject domain of first or foreign language learning, the system should have enough linguistic and meta-linguistic knowledge to correct the answers of the student, and to diagnose and explain his or her errors.

(iv) The computer as an *author and editor*. Intelligent word processors (author systems, author environments) will differ from present-day text editors by the inclusion of linguistic knowledge. This knowledge may be applied in functions like spelling and style checking and advice, on-line dictionaries, and in various additional text preparation aids.

(v) The computer as a *linguistic research assistant*. Rule testing devices can be built to test the adequacy of existing linguistic theories or to help in the construction of such theories. As we see this as one of the most important contributions of language technology, we will go into it in somewhat more detail in the next section.

Many of these applications will be treated more extensively in the remainder of this dissertation: rule testing devices in the next section and in Chapter 8, CAI in Chapter 7, intelligent word processing in Chapters 5 and 6, and speech interfaces in Chapter 8.

1.2.2. Linguistic Research Tools

An important achievement of language technology is the development of programs to test the adequacy of existing linguistic theories. During the design and the implementation of the program, inconsistencies, shortcomings, redundancies and vagueness (intensional inadequacy) inevitably come to light. An analysis of the nature of these shortcomings (they may be reparable or not) may lead to a modification of the original theory, or even to its rejection. Furthermore, once a theory has been implemented, it can be quickly and easily tested on a large corpus of 'real-life' natural language data as opposed to the selected example sentences common in theoretical linguistics.

The amount of work done in this direction is not very large, although its beneficial influence has often been attested (a.o. Brandt Corstius, 1978[6]; Van Bakel,

1983, Hoenkamp, 1985). Computer programs have been used to evaluate transformational grammars (Friedman, 1971), Montague grammar (Janssen, 1976) and Dik's functional grammar (Kwee, 1986).

If computer models can be used profitably to test existing linguistic theories, they can also be used to develop new linguistic theories (e.g. Gazdar, 1985; Thompson, 1983). Computer models have a distinct *heuristic function*, i.e. they can suggest ideas to the researcher through trial and error (Kempen, 1983), and they can help in overcoming the complexity barrier resulting from the application of large sets of interacting rules. Functional Unification Grammar (Kay, 1985) and GPSG (Gazdar, 1983) are examples of theories whose development was guided to a large extent by computer modeling. In Chapter 7, a tool for phonological research will be presented which illustrates the advantages of a computational model for linguistic research.

The use of computer programs in the testing and development of linguistic theories leads to a reflection on the relation between program and theory. Although programs may be developed which *implement* theories, the two should not be equated. Theories are sets of propositions while programs are sets of instructions (cf. Kempen, 1983). We interpret programs as *notations* for theories, much like the rule formalism in generative linguistics. We can prove that a program is a correct notation for a theory (if it comes up to the specifications of the theory when it is run), but this does not prove that the theory is correct. A program becomes a theory only when it is assigned an interpretation. E.g., we can construct a program which defines a set of linguistic rules. The program obtains a theoretical status only when (psycho-)linguistic relevance is claimed for these rules. Similarly, a program can use a number of levels of representation, but only when these are interpreted in some linguistic or psychological sense, they have theoretical status.

In a sense, there is no difference between using a paper-and-pencil notation to formulate a theory or using a computer program. However, programs have some unique properties: (1) they are formal and explicit to the extreme, (2) they are executable; i.e. they can give a physical interpretation to the concepts of a theory (it is precisely this property which makes them ideally suited to test the coherence of theories), and (3) they can be put to practical use, which gives them an economical

[6] Brandt Corstius even devoted one of his three laws of computational linguistics to it: *Every linguistic description, however exact, but not a program itself, turns out to contain an error if one tries to make a program of it* (translation, WD).

value.

1.3 Conclusion

Language technology is the part of computational linguistics which is concerned with the construction of computational models and their adaptation for practical applications. Computational models should be based on a computational theory, i.e. an abstract theory of natural language processing. Such a theory can be founded in psychology, linguistics and possibly neurology. The construction and testing of applications can provide valuable feedback for the organisation of such a theory, and through it for the psychological and linguistic theories on which it was based. Figure 3 pictures this view.

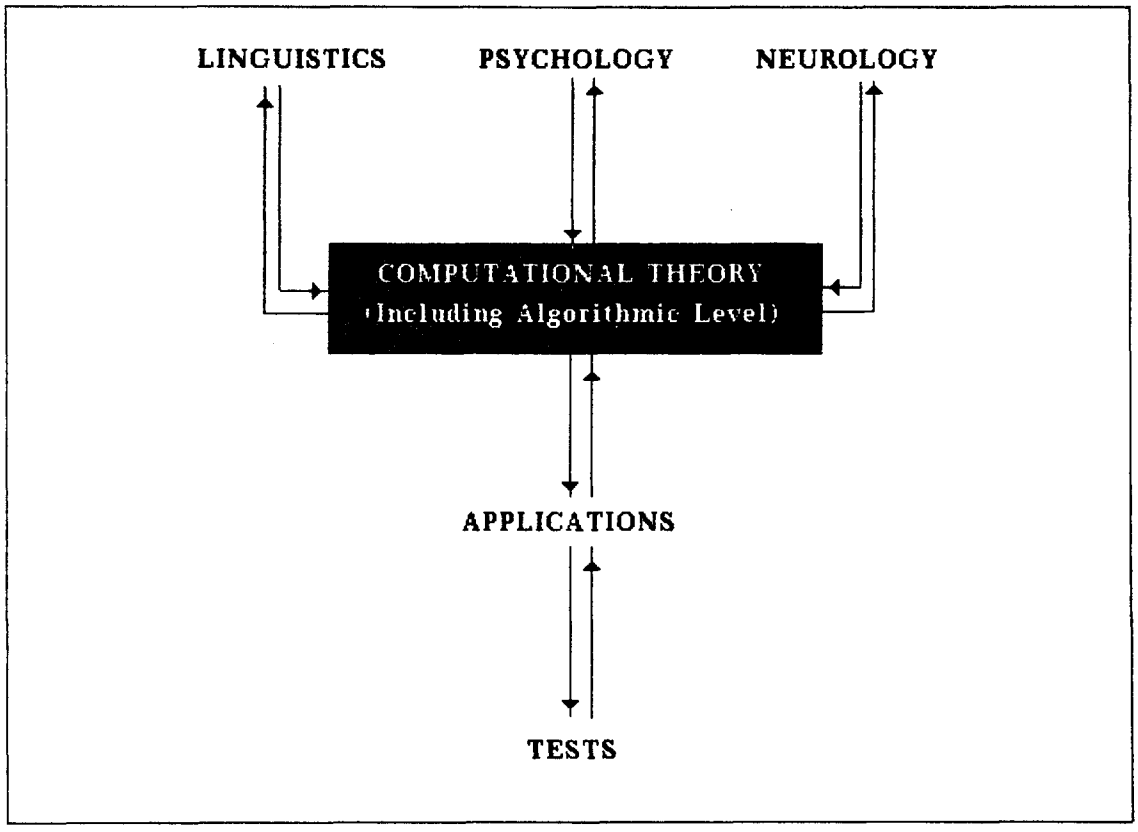


Figure 3. Language Technology. The downward arrows indicate the relation of determination, the upward arrows denote feedback relations.

CHAPTER 2

The Object-Oriented Programming Paradigm

Recently, computer science has seen the advent of a large number of new programming paradigms. The traditional *imperative* and *procedure-oriented* programming style is now being challenged by *logic-based, access-oriented, rule-based* and *constraint-based programming*. Another new development is *object-oriented programming*. After introducing the basic philosophy of this programming paradigm, its advantages and some variations between different object-oriented languages in sections 2.1 and 2.2, we will go into the syntax and semantics of the knowledge representation system KRS (Steels, 1985) in section 2.3. Finally, the usefulness of the object-oriented paradigm in linguistic knowledge representation and linguistic processing will be discussed in section 2.4.

2.1 Principles of Object-Oriented Programming[7]

> *Designing a good representation is often the key to turning hard problems into simple ones.*
> P.H. Winston

Objects are computational entities representing a concept in a domain of discourse. Objects in the domain of natural language processing could be *NP, direct object, syllable, focus*. Objects in the domain of office automation could be *letter,*

[7] A recent overview of the object-oriented paradigm can be found in Stefik and Bobrow (1986). The August, 1986 issue of Byte features a number of introductory articles on object-oriented programming, which seems to indicate that the paradigm is rapidly becoming popular.

communication-medium, invoice, employee. An object has an internal representation of the information associated with it, i.e. of its properties and behaviour. Declarative knowledge about an object (data) is represented by features (attribute-value pairs), and procedural knowledge by attached procedures (parameterised functions which can change the internal state of the object itself, or of other objects, or have some useful side-effect). The latter are often called *methods*.

A predominant metaphor used to describe action in an object-oriented system is *message-passing*: objects sending messages to other objects. Messages can set a property of an object to some value, retrieve the value of a property of an object or make an object execute a method (which may have arguments). E.g. an object *ORDER-55* could have an internal structure as in Figure 1.

```
object ORDER-55

    ordered-what: Lisp-Machine
    ordered-amount: 6
    ordered-who: Willy Wortel
    ordered-where: Symbolics
    communication-channel: Letter

    place-order (date):
        A procedure with as argument the date on which the order must
        be sent. When this specified date is equal to the present
        date, a message write is sent to an object
        letter, with arguments ordered-what, ordered amount
        etc..
```

Figure 1. Example of an object's internal structure (simplified).

The object ORDER-55 has some declarative information associated with it: *ordered-what, ordered-amount, ordered-who, ordered-where* and *communication-channel*. A message could be sent to ORDER-55 asking for the current value of one of these features. There is also one attached procedure,[8] *place-order* with one argument, *date*. Notice that the execution of this procedure causes ORDER-55 to send another message to another object (LETTER) with some arguments. In order to provide arguments for this message, the object has to examine its own internal state. We imagine the LETTER object to have an attached procedure *write* which fills out

[8] Procedures in most object-oriented languages are written in the programming language on top of which the object-oriented language was built, e.g. Lisp or Pascal. In this text, we shall use either a verbal description of what the procedure is supposed to do or an algorithmic description in 'formal English'.

the template of a standard order letter with the arguments provided whenever it is called.

The way information is computed depends on the object to which the message is sent. Another object ORDER-56 could have its own *place-order* method attached to it, resulting in different behaviour if the same message is sent. This is sometimes called *polymorphism*. In this respect, object-oriented languages differ radically from procedure-oriented languages (also called action-oriented or function-oriented languages), which take a procedure as central. E.g. a generic[9] function + would be implemented as shown in Figure 2(i) in a procedure-oriented and as in Figure 2(ii) in an object-oriented language.

```
define function +, with arguments n and m

    if n or m is real
    then execute procedure real-plus

    if n and m are integers
    then execute procedure integer-plus

    if n and m are strings
    then execute procedure string-concatenate

    else signal argument error
```

Figure 2(i). Function-oriented implementation of function +.

```
object REAL
    method: + (arg)
 procedure real-plus

object INTEGER
    method: + (arg)
 procedure integer-plus

object STRING
    method: + (arg)
 procedure string-concatenate
```

Figure 2(ii). Object-oriented implementation of function +.

In a function-oriented approach, a generic function is accessed through its name, and a type-check on the arguments is performed to determine which sub-procedure is

[9] Generic functions are functions that apply to more than one data type.

to be used. In an object-oriented approach, the different sub-procedures are associated directly with the different data-types, which are implemented as objects.

Different objects are related to each other through hierarchical links. Mostly, more specific objects are defined as sub-types of more general types, but it is preferable to understand *type hierarchies* in a purely technical sense, without confusing types with categories (classes) or species (cp. Steels, 1985), as this may lead to mistakes[10]. Part of a type-hierarchy in the domain of office systems could be the one in Figure 3.

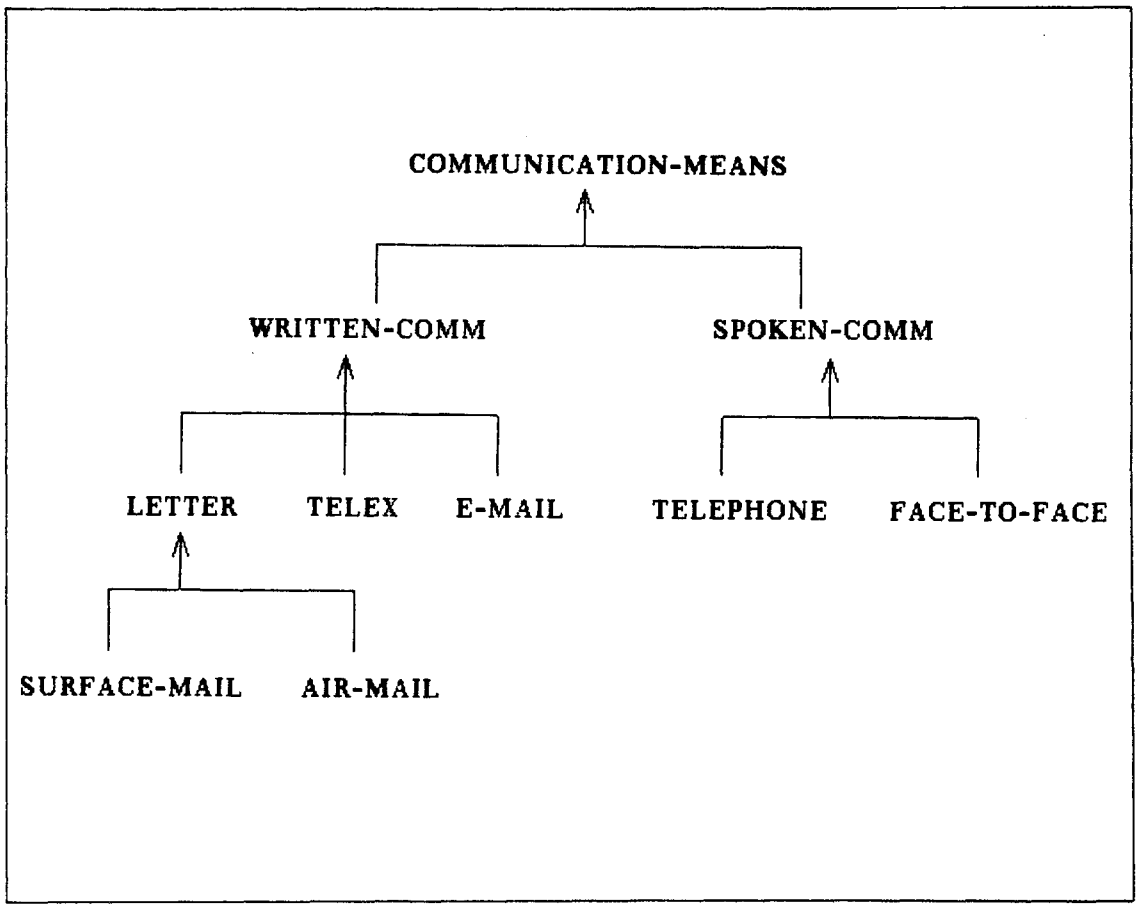


Figure 3. Part of a hypothetical hierarchy in an office system.

[10] Consider for example a hierarchical link between *Clarence* and *lion*: Clarence is a lion. And *lion* has the attribute-value pair (extinct false) associated with it. Clarence would inherit this feature although only species can be extinct, and not individuals (except in a metaphorical sense).

Information present in types is available to sub-types through a process called *inheritance*. The object SURFACE-MAIL can inherit information from the types it belongs to (LETTER, WRITTEN-COMMUNICATION, COMMUNICATION-MEANS). Inheritance reduces redundancy: if two objects are almost alike, one can be made a sub-type of the other, or both can be made sub-types of a third. Only those portions of the knowledge associated with an object which are different from its parent objects need be stored. Inherited information is overruled (substituted) if the information is provided in the object itself (Figure 4).

```
object PERSON
  number-of-legs: 2
  number-of-arms: 2
  is-rational: yes

object JOHN
  type: PERSON

object IRRATIONAL-PERSON
  type: PERSON
  is-rational: no
```

Figure 4. Example objects for PERSON, JOHN and IRRATIONAL-PERSON.

The object JOHN inherits all information associated with PERSON, without a need to store it in John. IRRATIONAL-PERSON inherits all information associated with PERSON except the property *is-rational* which is overridden by the information associated with IRRATIONAL-PERSON.

Family relationship terminology is used to describe the relations among objects: the types an object inherits from are the *parents* of that object, inheriting objects are *children* or *inheritors*, parents of parents of an object are *ancestors*.

When an object inherits from more than one type (*multiple inheritance*), the inherited properties are combined in some pre-defined way (e.g. union, with elimination of duplicate features). If different parents of an object have the same features or methods, it is a search algorithm (breadth-first or depth-first, left-to-right or right-to-left) which determines which version is kept, and which one is destroyed as a duplicate. Thus, in the following (tangled) hierarchy (Figure 5), the search algorithm determines which version of the *life-expectancy* property is inherited by BERT. In a breadth-first search, BERT inherits a low life expectancy, in a depth-first search a high one. Through multiple inheritance, existing pieces of knowledge can be combined into more complex wholes (*object-composition*). In our example, BERT is

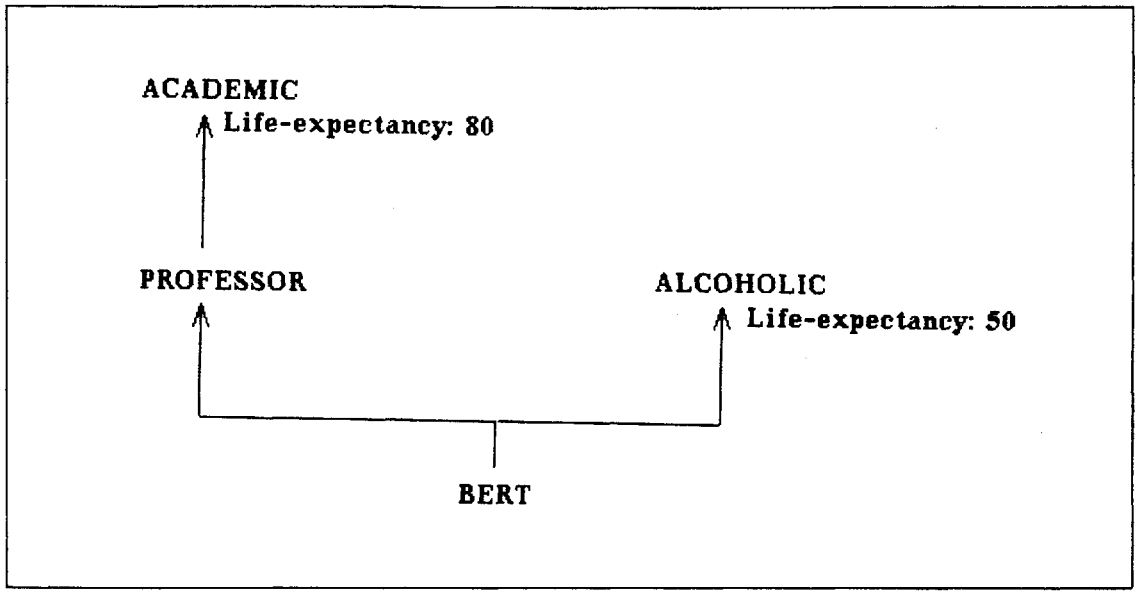


Figure 5. A tangled hierarchy.

composed as an alcoholic professor.

Apart from the way the inheritance hierarchy is searched, decisions must be made about the way the inherited material is combined. Often, a procedure is inherited as a monolithic whole. Some languages, however, provide utilities for the inheritor to modify the code of the procedure before executing it. This can be done either by directly grabbing the definition of the procedure through an operator, change it, and evaluate the result (as in the language CommonORBIT, De Smedt 1987); or by providing demons which add pieces of code before and after the main code of a method at compile time, plus a declarative language which can be used to change the default behaviour of these demons (as in FLAVORS, Weinreb and Moon, 1981). Both approaches result in an ability to combine different methods. It would be possible, for example, in the example of Figure 5 to return the average of the life expectancies for academics and alcoholics as the life expectancy of Bert.

To what extent are object-oriented languages better than traditional programming languages? For now we will concentrate on general advantages. Later we will point out their relevance to linguistics.

(i) *Modularity* (encapsulation): No knowledge about the internal details of an object type (data type) is needed. The actual implementation of an object can be changed radically (e.g. from a property list to an array), without any difference to the user. Furthermore, when adding new types, existing objects need not be

redefined. Internal details can be changed without affecting other objects. An object may be imagined as a black box which accepts a number of inputs and produces a number of outputs, or as an expert who can answer a number of questions (stored in its properties), and who can solve some problems (by applying his methods).

Due to inheritance, information can be changed in one place, and inheritors inherit this change automatically (if provision is made for this). As real-life applications typically involve thousands of objects, the time saved by this cannot be overestimated. Thus, *extensibility* and *easy modification* ensure the modularity needed in the development and maintenance of large programs. A term currently in vogue for describing re-usable software modules which are largely independent from a particular application is *Software ICs* (Ledbetter and Cox, 1985).

(ii) *Style*: Object-oriented systems are conceptually simple to work with. Most of us think and reason about a problem in terms of entities, properties of entities, relations between entities and actions which are natural for these entities. An almost one-to-one mapping of these conceptual units is possible with the computational units of an object-oriented language. Thinking about an algorithm in terms of objects makes it easier to understand. This close relation between the way we think and the way the program is structured, makes the translation of the abstract solution to a problem into program code easier, and consequently less error-prone. Furthermore, all information about an object is collected in one place. Due to modularity and the simplicity of syntax and semantics of most object-oriented languages, programs are well-structured and easy to read.

(iii) *Efficiency*: Efficiency of an object-oriented system depends on the number of objects and generic procedures in a particular application, and on a number of design features. Object-oriented systems are efficient when lots of objects are needed. Inheritance by search (in which information is looked for in the type hierarchy every time it is needed) prevents that copies of the same function have to be stored in too many objects, thereby keeping access times relatively low and storage requisites reasonable. Inheritance by copying (in which information associated with the parent is copied to the children when they are created) reduces search time in the hierarchy, but increases storage overhead, and an explicit recomputation in all inheritors is needed whenever changes are made in a type (*consistency maintenance*).

Disadvantages of object-oriented systems sometimes pointed at are, first of all, that when adding a new function, it (or a variant) must be added to all relevant types, which may require a restructuring of the object network. Second, information about one function is scattered over different objects (whereas in a function-oriented approach, information about one object is scattered over different functions). And finally, careful thinking is required about which information should be represented as objects, and which as properties, and how to structure the inheritance network. Often, different options are open to the programmer, one of which turns out to be preferable, but not necessarily *before* programming has begun. Object-oriented programming mechanisms may be too powerful in some cases.

Summing up, an object-oriented approach is ideal for systems in which it is necessary to represent large quantities of interacting entities. The approach has been successfully applied to graphics (window systems), natural language semantics, VLSI-design, expert system development, etc.

2.2 An Overview of Object-Oriented Systems

Message-passing between computational entities as a programming paradigm has been developed at M.I.T. and XEROX in the early seventies. Carl Hewitt and collaborators (1973) developed an actor formalism based on message passing, later implemented as the ACT1 language (Lieberman, 1981). Alan Kay (1974) initiated work on SMALLTALK, which incorporates a similar philosophy. Inheritance derives from the work in semantic network theory which was started by Ross Quillian (1968). Attempts to connect related nodes of a network into a more coherent whole (*partitioned networks*, Hendrix, 1979) are more or less equivalent with the computational object idea. *Frame theory* (Minsky, 1975; Kuipers, 1975; Winograd, 1975; Metzing, 1979) has certainly influenced a lot of object-oriented concepts and languages (Steels, 1981b). Frame theory is a psychological theory about the organisation of human memory. Frames are complex, partially declarative, partially procedural representational structures which hold all information about a limited subject matter. They can be interpreted as experts in a small domain, or as structures representing a stereotyped chunk of knowledge (a prototype). The internal structure of a frame is a set of named slots which are filled by other frames or by identifiers. Fillers can have restrictions (type checking), defaults (values to be used if no explicit value is provided), demons (actions to be performed before or after a slot is filled) and meta-knowledge attached to them. Larger frames, specifically designed for representing sequences of

events, are *scripts* (Schank and Abelson, 1977). Frames and scripts can be easily implemented in most object-oriented languages. The language KRL (Bobrow and Winograd, 1977) and the first versions of ORBIT (Steels, 1981b) are examples of implementations of the frame idea.

Differences and variations in the way programming languages implement the object-oriented philosophy described above abound. Most object-oriented languages are written on top of existing programming languages (Lisp: CommonORBIT, FLAVORS, KRS, ExperCommonLISP; Pascal: OBJECT PASCAL, CLASCAL; Algol: SIMULA; Forth: NEON), others are not parasitic upon an existing language (SMALLTALK). Some have a specific syntax for message-passing (FLAVORS, KRS), others represent function application and message-passing in a uniform way (CommonORBIT). Often a distinction is made between object-types and object-instances. E.g. in FLAVORS, types (called Flavors) cannot receive messages, and instances can have properties (instance variables), but no associated methods; the flavors act as a mould to the instances. Other languages do not make this difference (KRS, CommonORBIT). An object can be implemented in a variety of ways: as a dynamic record, as a property-list, or as a function.

Some object-oriented languages support inheritance in tangled networks (CommonORBIT, FLAVORS), others do not (KRS, at least not by default). We have already mentioned the difference in the way the inheritance hierarchy is searched if multiple inheritance is supported (depth-first in FLAVORS, breadth-first in CommonORBIT). Also, the way inheritance is implemented may vary: by copying the information of a parent to its children, or by searching it each time it is needed. Related to this choice is the presence or absence of a mechanism to propagate changes in a parent to its children. This propagation (*dynamic inheritance* or *delegation*) is essential for modularity. Some object-oriented languages provide back-pointers or bi-directionality (e.g. in CommonORBIT, if an object x has a property p, inherited or otherwise, then x can be retrieved through p), others do not (e.g., KRS, FLAVORS).

SMALLTALK, the object-oriented language most widely used is described in Goldberg and Robson (1983). FLAVORS is documented in Weinreb and Moon (1981) and Cannon (1982), ORBIT and CommonORBIT in Steels (1981a, 1981b, 1982) and De Smedt (1984, 1987) and KRS in Steels (1985) and Van Marcke (1987).

2.3 Syntax and Semantics of the KRS Concept System

A new language brings with it a new model of the machine.
Lawrence G. Tesler

KRS (Steels, 1985; Van Marcke, 1987) was designed to be able to incorporate and integrate different formalisms (functional, network, rules, frames, predicate logic etc.) into a single system. It is also possible to implement new formalisms on top of KRS. However, in the context of this dissertation, we will interpret it as a frame-based object-oriented language for knowledge representation. In the course of implementing the linguistic knowledge described in Part II, we experimented with several object-oriented programming languages (ORBIT, FLAVORS and KRS). KRS seemed to us the most versatile and powerful implementation medium. All examples in following chapters will be written in a kind of 'stylised KRS' (with a simplification of syntax for readability). At this point we will give only a short introduction to part of the KRS concept system. More detailed information will be given in the following chapters whenever it is relevant.

Knowledge is represented by means of *descriptions* (something which stands for something else). E.g. *Human(Socrates)* is a description (in this case in predicate calculus) describing a state of affairs in a world. Note that with this definition, natural language, too, is a knowledge representation system. Phrases and sentences are descriptions of (real or imaginary) states of affairs in a possible world (cp. Droste, 1985).

In KRS, descriptions are called *concepts*. Concepts in KRS are the same as objects in other object-oriented languages (see section 2.1). This may cause some confusion. In the remainder of this text, we shall use both to mean the same thing. A concept has a *name* and a *concept structure*. A concept structure is a list of *subjects* (slots), used to associate declarative and procedural knowledge with a concept. A concept name is only a mnemonic label, meaningless to the system. Subjects are also implemented as concepts, which leads to a uniform representation of objects and their associated information. The filler of a slot is called the *referent*. Concepts can be defined with a function *defconcept* and subjects with a function *defsubject*. Information is retrieved from concepts by means of *definite descriptions*. In Figure 6 a few examples are given.

```
(DEFCONCEPT CHARLOTTE
   (HUSBAND HUYBERT)
   (CHILD CASPER)
   (LIVES-IN NIJMEGEN))

A concept Charlotte is defined with a concept structure (husband Huybert)
(child Casper)(lives-in Nijmegen). Each of these lists is a subject with an
access (e.g. child) and a referent (e.g. Casper). Note that a concept
description is not a definition of the concept in some theoretical sense;
the concept structure of Charlotte is not a definition of Charlotte, but
some information associated with this concept in a particular context.

(DEFSUBJECT HOME OF CHARLOTTE CITY-HALL)

This adds a subject home with referent City-hall to the concept structure
of the concept Charlotte.

(>> CHILD OF CHARLOTTE)  -->  <CASPER>

This is an example of a definite expression, it returns the referent concept
of the subject with the access child associated with the concept
identified by Charlotte. It is roughly comparable with message-passing
in other languages.

(DEFCONCEPT CASPER
    (MOTHER CHARLOTTE)
    (HAIR-COLOUR WHITE))

(>> HAIR-COLOUR CHILD OF CHARLOTTE)  -->  <WHITE>

As the referent of a subject is itself a concept, accesses via paths are
possible as well. The definite description is equivalent to

(>> HAIR-COLOUR OF
      (>> CHILD OF CHARLOTTE))

which is equal to

(>> HAIR-COLOUR OF CASPER)  -->  <WHITE>
```

Figure 6. KRS Examples.

Inheritance — the process of looking up information in the parents of a concept if it is not defined in the concept itself — is single by default in KRS, i.e. each concept can have only one type or parent (but a type can have many instances or specialisations). This default inheritance system can be changed by the user, however, for example into a multiple inheritance scheme. A concept is made an inheritor of a type by means of an *indefinite description*. Due to the fact that referents of subjects are concepts, inheritance works through subjects as well. Some inheritance examples are listed in Figure 7.

The traditional language philosophical distinction between *extension* and *intension* is adopted in KRS. In linguistic semantics, proper names refer to entities (Fido ->

```
            (DEFCONCEPT PERSON
                (NUMBER-OF-LEGS TWO)
                (RATIONAL YES))

            (DEFCONCEPT IRRATIONAL-PERSON
               (A PERSON
                (RATIONAL NO)))

            (DEFCONCEPT JOHN
              (AN IRRATIONAL-PERSON))

This KRS code defines a type relation between PERSON, IRRATIONAL-PERSON,
and JOHN by means of indefinite descriptions: (a(n) <concept-description>)

            (>> NUMBER-OF-LEGS OF JOHN)   -->  <TWO>

This definite description looks for the subject NUMBER-OF-LEGS, first in
JOHN, then in IRRATIONAL-PERSON, and finally in PERSON, where the subject is
found.

            (DEFCONCEPT JOHN
              (A PERSON
                (AGE (A NUMBER))
                (FATHER (A PERSON))))

            (>> NUMBER-OF-LEGS FATHER OF JOHN)  -->  <TWO>

This example illustrates inheritance through subjects.
```

Figure 7. More KRS Examples.

FIDO), predicates refer to sets of entities (dog -> the set of all dogs), and the extension of a sentence is its truth value. The intension of proper names and predicates is, depending on the theory, either a conceptual structure or a defining property (the necessary and sufficient conditions to fall within the extension). The intension of a sentence is a set of truth conditions (necessary and sufficient conditions for the sentence to be true). Intension is a linguistic notion (defined as a set of relations with other linguistic expressions) as opposed to extension, which relates language to the world. Extensions are taken to be contextually determined, intensions are considered constant. It is therefore possible to define the intension as a function which yields the extension when applied to a particular context.

A KRS concept is linked to its extension by means of a special subject, called the *referent*. E.g. the referent (or extension) of a concept *two* is a Lisp number 2. The referent of a formula (a program fragment) is a piece of Lisp code. The same referent can have different descriptions. For example, the concepts <Number-2>, <two> and <twee> all have as a referent Lisp number 2. A special notation exists for describing the referent of these *data-concepts* (a category of concepts

having Lisp data-types as their referent): e.g. [number 2] is a shorthand for (A NUMBER (REFERENT 2)). Concepts which are not data-concepts have other concepts or an abstraction as their referent. Only data-concepts have 'physical' referents (as defined by some action or state in the machine through Lisp data structures). This implies that most reasoning is done at the abstract level of definitions and descriptions, as most referents of concepts in an application cannot be represented in the machine (you cannot put a dog into the computer).

Apart from a referent, concepts can have a subject *definition* (intension). A definition is a function which computes the referent of a concept in a particular context (a Lisp environment). The referent of a definition must be an executable Lisp form. A basic feature of KRS is therefore that *the referent of a description is the evaluation (execution) of the referent of the definition of the description* (Figure 8).

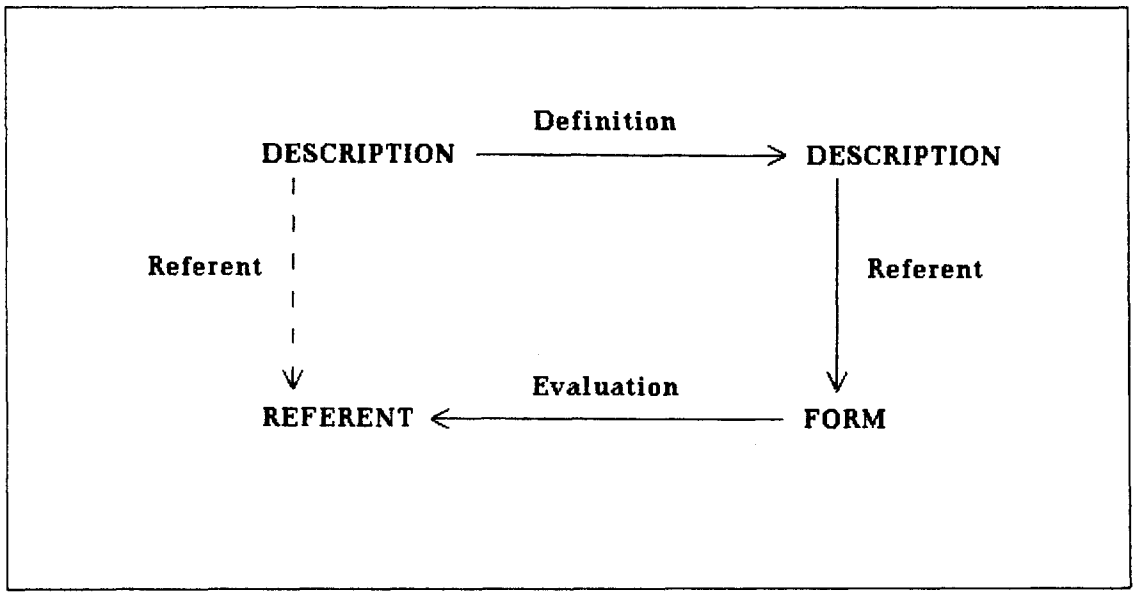


Figure 8. Referent computation in KRS.

In Figure 9, a (simplified) BNF summary of the syntax of KRS is given.

This brief sketch of the object-oriented programming system KRS suffices to follow the representation of linguistic knowledge in Chapters 3 and 4. Although we will not go into this in the present dissertation, it is clear that KRS can be straightforwardly adapted to implement logic grammars such as the one developed by Montague (1974) due to its explicit representation of extensional and intensional meaning.

```
CONCEPT-DESCRIPTION     := CONCEPT-NAME | DEFINITE-DESCRIPTION | CONCEPT-
                           STRUCTURE | INDEFINITE-DESCRIPTION
CONCEPT-NAME            := SYMBOL
DEFINITE-DESCRIPTION    := (>> ACCESS-1 ... ACCESS-n OF CONCEPT-DESCRIPTION)
INDEFINITE-DESCRIPTION  := (A CONCEPT-DESCRIPTION CONCEPT-STRUCTURE)
CONCEPT-DEFINITION      := (DEFCONCEPT CONCEPT-NAME CONCEPT-DESCRIPTION)
SUBJECT-DEFINITION      := (DEFSUBJECT ACCESS OF CONCEPT-DESCRIPTION
                            CONCEPT-DESCRIPTION)
CONCEPT-STRUCTURE       := (ACCESS-1 CONCEPT-DESCRIPTION)
                           ...
                           (ACCESS-n CONCEPT-DESCRIPTION)
ACCESS-i                := SYMBOL
```

Figure 9. Simplified KRS summary.

2.4 Object-Oriented Computational Linguistics

> *Some kinds of notation seem to fit the sort of facts one encounters in some domain; others, which may ultimately be equivalent in some sense to the former kinds, do not.*
> Gerald Gazdar

In this section we will try to show that an object-oriented language is notationally adequate for the description of linguistic knowledge and processes. This demonstration can only be theoretical, and somewhat intuitive. Much of the power of object-oriented programming becomes obvious only through experience and by programming the same problem in both object-oriented and alternative ways. By means of examples, Part II of this dissertation will give more substance to the claims made here.

We are fully aware that all programming paradigms are equivalent in that the programs they generate are reduced ultimately to machine language (i.e. they are weakly equivalent to Turing machines). But to the programmer (and to the computational linguist) they are different in the metaphors they provide to conceptualise the problem domain, and the image they generate of the machine. In that sense, the particular programming paradigm used has a distinct effect, not only on the speed and ease of theory building, but on the theory building itself.

Besides the general computational advantages of the object-oriented paradigm outlined in section 2.1 (modularity, ease of programming, clarity of style and efficiency), we see five specific advantages for linguistics.

(i) *Hierarchies* are a powerful mechanism for describing generalisations in an elegant and effective way, and as briefly as possible. Generalisation can be achieved by attaching general information (defaults) to concepts from which a number of other concepts inherit. Afterwards, one only has to specify in what respects an inheritor differs from a parent. This approach not only assures an efficient implementation (in terms of storage), it also brings conceptual clarity about the relations between theoretical concepts.

(ii) The dichotomy between *rules and exceptions* is nicely captured by overruling inherited defaults. We can state, for example, that all vowels are stressable by associating this information to the concept vowel. The exceptional status of *schwa* can then be shown by letting it inherit from the type vowel, but overruling the subject *stressable*. Complexity of a language can be operationalised as a function of the number and kind of 'exceptional' specialisations (instances) of 'regular' concepts.

(iii) Related to (ii), the fuzziness of linguistic theoretical concepts can be modeled by multiple inheritance (semi-vowels inherit from both vowels and consonants, nominalised verbs from both nouns and verbs) or by specialisation (a new concept which inherits all but a few of the subjects of its parent). That way, stereotypes (prototypes) can be naturally and easily modeled.

(iv) All knowledge which plays a role in linguistic processes (world, situational and linguistic knowledge) and all levels of linguistic description (from the pragmatic to the phonetic) can be represented in a simple and uniform way (objects, inheritance hierarchies, methods and features). Even meta-linguistic knowledge can be represented[11]. Furthermore, it seems natural for most linguistic theories to use a formalism based on graphs (nodes with features and arcs defining relations between nodes). Case frames, phrase structure trees, ATNs, scripts, semantic networks and functional descriptions are some instances of this kind of formalism. They can all be represented straightforwardly in an object-oriented programming style. Nodes map to objects, features of nodes to features of objects, relations between nodes to features of objects or hierarchical relationships. This

[11] In a sense, most linguistic knowledge represented in the following chapters *is* meta-linguistic knowledge. For instance, KRS concepts exist for morphological boundaries, particular words, syntactic categories, dictionaries, etc. and for the relations between these concepts. Describing this KRS knowledge base in natural language produces metalinguistic sentences: *'Table' is a noun, 'z' is a fricative, A phonological rule consists of a condition part and an action part.* (a meta-rule) (cp. Droste, 1983).

shows that an object-oriented notation 'is close to the sort of facts one encounters in the linguistic domain' (see the citation by Gazdar at the beginning of this section).

(v) Language processing (generating and interpreting) can be viewed as a problem-solving task using hierarchies of linguistic knowledge concepts. Complex rule interactions and sequences of actions can be formulated relatively simply. By looking for an optimal division of knowledge among concepts, and an optimal explication of the interrelations between them, complicated actions can be described in a simple, intuitively plausible way.

However, the expressive power of the object-oriented paradigm can also be regarded as a disadvantage: there are almost no restrictions on the way knowledge can be defined, organised and used. Different alternatives can be considered in the organisation of hierarchies and inheritance, different places in the hierarchy are equally defendable for the attachment of procedures.

Relative to the methodological criteria which are important to us, however, these dilemmas can be resolved in different ways. If we want to simulate human verbal behaviour, our freedom in arranging knowledge and procedures will be restricted by psycholinguistic experimental data. If it is linguistic adequacy we are after, the system would have to be organised so as to adhere most closely to linguistic methodological criteria such as maximal generalisation. Finally, if we have a commercial application in mind, the construction process must be guided by considerations of computational efficiency. In Chapter 1 it was claimed that a computational theory of verbal behaviour should incorporate all these different constraints into a coherent whole.

In Part II of this dissertation, a KRS implementation of a computational model of aspects of Dutch phonology and morphology will be developed. A frame-based or object-oriented approach has been used extensively in the implementation of world knowledge for use in semantic interpretation. Similar approaches to phonology and morphology are less frequent.

PART II

LINGUISTIC KNOWLEDGE REPRESENTATION AND PROCESSING

In the preceding chapters, language technology was defined as the design of applications of a computational linguistic model. Object-oriented programming was put forward as an ideal paradigm for the programming of linguistic knowledge and processes.

In the following chapters, an object-oriented implementation of aspects of Dutch morphology and phonology will be described. The description of a complete model of Dutch morpho-phonology would involve a text several times the size of the present one. We will restrict our attention to the synthesis of verbal inflections, the analysis of compounds and the detection of internal word boundaries (Chapter 3), and syllabification and phonemisation algorithms (Chapter 4). Special emphasis will be put on the interaction between morphological and phonological knowledge and processes, and on the extensibility of the knowledge base developed. Also, a principled study of the interaction between spelling, phonological and lexical representations will be made, and some considerations in the design of a lexical database will be put forward.

CHAPTER 3

Aspects Of Dutch Morphology

In this chapter[1], a knowledge base of morphological concepts and morphological processes operating on this knowledge base will be built up. The resulting model, which will be further extended with phonological information in the next chapter, serves two aims: first, it should solve the *technological problem*; i.e. the model will have to be useful for the different applications we had in mind for it, and second, it should be constrained by linguistic and psychological evidence. As such, it comes up to the methodological requirements put forward in Chapter 1 of this dissertation.

Morphology is a branch of linguistics which studies the internal structure of existing complex (polymorphematic) words and the rules with which new complex words can be formed. We will adopt a *lexicalist* framework[2] of word formation here, presupposing the existence of an independent word formation component, the rules of which operate before lexical insertion. Our model will not be orthodoxically lexicalist, however, as we will allow all complex words to be entered in the word list, and not only those with at least one idiosyncratic property.

In *computational* morphology, the morphological capacities people exhibit (producing and interpreting morphologically complex words) are simulated. Aspects of both synthesis (production or generation, section 3.1) and analysis (interpretation or recognition, section 3.2) will be treated, as well as the role of the lexicon (section 3.3) and word formation rules in both processes.

---

[1] This chapter is partly based on Daelemans (1986).

[2] See e.g. Chomsky, 1970; Halle, 1973; Jackendoff, 1975; Aronoff, 1976; Booij, 1977 and others.

In our model, morphological synthesis and analysis will be regarded as fundamentally different processes. Synthesis (the transformation of a lexical representation into a phonetic or orthographic representation) is determined uniquely by the properties of the lexical representation. Analysis on the other hand (assigning a lexical representation to the spelling or sound representation of a word form), is determined not only by the input representation, but also by linguistic and extra-linguistic context. The reason for this is that different underlying representations can be realised as the same sound or spelling image, introducing ambiguity. Furthermore, both processes access the lexicon in essentially different ways. Synthesis enters the lexicon with a lexical representation, analysis with a phoneme or grapheme representation. As far as possible, however, the same knowledge base will be used by both processes.

The model developed will be compared to other computational models of morphology, and to results of psycholinguistic research (section 3.4). Only parts of the complete model, which is still under development, will be treated here, notably the synthesis of verbal inflections, the analysis of compounds, and the detection of internal word boundaries. We will not be concerned with the semantic level in the lexicon.

Before starting the discussion of our model, it may be useful to define the terminology which we will use in this and subsequent chapters, since much confusion exists in the literature. Our definitions will be based mainly on Matthews (1974) and Lyons (1981).

Morphemes are the smallest elements of the language with a semantic or syntactic function (roots, stems and affixes). A *word form* is a morpheme or a combination of morphemes. It consists of either a string of letters or a string of phonemes. Word forms are generated by *morphological processes*, captured in *morphological rules*. Dutch productive morphological processes are *affixation*, *compounding* and *compounding affixation*. There are also relics of an *Ablaut* process. A *paradigm* (*lexeme*, lexicon entry) is a set of formally related word forms which have the same *base form* (called *root* if it is unanalysable and *stem* if it is complex) but different *affixes*. In exceptional cases, a paradigm may have more than one base form (this is called *suppletion*). E.g., the paradigm of the verb *kopen* (to buy) has base forms *koop* and *kocht*. One, privileged, word form in a paradigm is the *citation form*, e.g. the infinitive of verbs and the singular of nouns. The citation form is used to name

the paradigm. We can say e.g. that the paradigm or lexeme *lopen* consists of the word forms *lopen, loop, loopt, lopend, gelopen, liep, liepen, lope* with *lopen* as the citation form.

It may also be useful to describe the different kinds of word level ambiguity which are usually distinguished. *Class ambiguity* exists when a word form can belong to two different syntactic classes. E.g. *loop* can be either a noun (*barrel*), or a verb (first person singular present indicative of *lopen* (to run). Class ambiguity can exist between paradigms (as in the case of *loop*) or within a paradigm (*subclass ambiguity*). E.g. *lopen* can be both infinitive, and plural present indicative. *Semantic ambiguity* can be due to either *homonymy* (unrelated words have the same spelling or pronunciation e.g. *bank* as *a financial institution* and as *ground near the river*), or *polysemy* (related words with different meanings e.g. *dry* as *not wet* and as *not sweet*, for wine). The distinction between homonymy and polysemy cannot always be easily made from a synchronic point of view.

3.1 Morphological Synthesis

In this section, part of a synthesis program written in KRS will be described. Synthesis programs can be applied as a module in the generation part of larger natural language processing systems, and in automatic dictionary construction (Chapter 8).

The task of the program fragment is to compute the inflections of Dutch verbs on the basis of their infinitive and it is intended to cover all regular and irregular categories of verbs. The inventory of verbs (the data) was taken from the authoritative *ANS* (Algemene Nederlandse Spraakkunst, General Grammar of Dutch, 1984).

The *levels of description* will be mainly the lexical and the spelling levels, but phonological information will be discussed whenever it plays a role in the computation of the spelling image of word forms. The spelling and phonological levels are carefully kept apart. An autonomous spelling level provides insight into the structure of the spelling system, describes the knowledge people must have in order to spell the various inflected word forms correctly, and may thus provide a scientific background to the evaluation of different proposals for spelling change. Similar ideas have been put forward by Zonneveld (1980), Kerstens (1981) and Wester (1985b). The phonological level will be described in Chapter 4. We have tried to exploit this modularity as far as possible in order to have a clear view on the precise interaction between both levels. In practice, this approach implies that phonological information is used to

compute word forms when it is available; but when it is absent, the system can nevertheless resort to other strategies and heuristics. This was necessary to make the knowledge base flexible enough to be useful in different applications. Input to our programs is mostly a spelling representation.

The *linguistic description model* adopted is essentially Item and Process (IP, see Matthews, 1974 for a discussion of various models). Abstract morphemes are the basic elements of morphological structure, and their combination in word formation happens by means of morphological processes which may modify morphemes. These modifications are reflected in the realisation (in spelling and sound) of the new word form. The introduction of the possibility of morphological processes modifying morphemes is an essential departure from the Item and Arrangement model (IA), and is necessary to describe non-sequential effects such as vowel changes. Both IP and IA are superior to the Word and Paradigm (WP) model in that they capture important linguistic generalisations instead of simply listing the inflected forms of a lexeme. However, the notion of paradigms acting as prototypical cases from which the inflections of other verbs can be induced (e.g. for learning and teaching purposes), is kept in our description. But our paradigms have a different meaning: they denote clusters of morphological processes rather than lists of examples. A WP-approach is also used to describe unsystematic irregularities (e.g. the suppletive verbs discussed earlier).

3.1.1 Objects in the Domain of Synthesis

When developing an object-oriented system, a set of concepts in the problem domain (morphological synthesis) must be defined, and their properties and interrelations made explicit. Figure 1 shows part of the object-hierarchy necessary to generate verbal inflections. We use normal KRS-syntax, but pieces of Lisp code will be paraphrased in 'formal English'. These parts will be set apart within curly brackets. Recall that in KRS a concept description (i.e. the subjects associated with it) is not the same as a *definition* of the described object in a particular theoretical framework. It is only a description of what is relevant in a particular context. To the concept description of morpheme, for example, subjects relating to phonological structure will be added later (Chapter 4). When incorporating a semantic level in the system, still other subjects will have to be added.

```
(DEFCONCEPT BOUNDARY
     (LEXICAL-REPRESENTATION (A STRING)))

(DEFCONCEPT WORD-BOUNDARY
     (A BOUNDARY
       (LEXICAL-REPRESENTATION [STRING "#"])))

(DEFCONCEPT MORPHEME-BOUNDARY
     (A BOUNDARY
       (LEXICAL-REPRESENTATION [STRING "+"])))

(DEFCONCEPT MORPHEME
     (LEXICAL-REPRESENTATION (A STRING))
     (BOUNDARY (A BOUNDARY)))

(DEFCONCEPT FREE-MORPHEME
     (A MORPHEME
       (BOUNDARY (A WORD-BOUNDARY))))

(DEFCONCEPT BOUND-MORPHEME
     (A MORPHEME))
```

Figure 1. KRS Implementation of some morphological concepts.

The knowledge expressed in Figure 1 might be paraphrased as follows. All *boundaries* have a lexical-representation. There are two sub-classes of the boundary type: *word* and *morpheme* boundaries. All instances of the former have lexical representation '#', instances of the latter have lexical representation '+'. The difference between both types plays a role in morphological analysis and in phonological processes such as syllabification (Chapter 4.1). We will argue in later chapters (Chapter 5 and 6) that the design of satisfying hyphenation and spelling/typing error detection algorithms for Dutch relies in large part on this distinction. We expect the *morpheme* object to have a lexical representation and a boundary. Whether a boundary follows or precedes a morpheme depends on its type. A sub-class relation with *free* and *bound morpheme* is made. All free morphemes have a word boundary. Figure 2 continues the description of objects in the morphological synthesis domain by means of KRS concepts. In Figure 2, *prefixes* and *suffixes* are classified as bound morphemes. They inherit the lexical representation and boundary subjects from their ancestors. All prefixes have a word-boundary. An affix knows how to append itself to a morpheme (the context); the procedure to do this is described in its *append* subject.

As an example, consider how the knowledge base built up so far can be used to compute the lexical representation of the past participle of the verb *werken* (to work). We compute the referent of the concept which results from 'appending' the relevant affixes to a string representing the root of a verb.

```
(DEFCONCEPT PREFIX
     (A BOUND-MORPHEME
       (BOUNDARY (A WORD-BOUNDARY))
       ((APPEND (?CONTEXT)
        (A STRING
            {Concatenate the following strings:
              (>> REFERENT LEXICAL-REPRESENTATION)
              (>> REFERENT LEXICAL-REPRESENTATION BOUNDARY)
              (>> REFERENT OF ?CONTEXT)}
           ))))

(DEFCONCEPT PAST-PARTICIPLE-PREFIX
     (A PREFIX
       (LEXICAL-REPRESENTATION [STRING "gə"])
       (BOUNDARY (A WORD-BOUNDARY))
 ((APPEND (?CONTEXT)
   (A STRING
    {Concatenate the following strings:
     (>> REFERENT LEXICAL-REPRESENTATION)
     (>> REFERENT LEXICAL-REPRESENTATION BOUNDARY)
     (>> REFERENT OF ?CONTEXT)}
      ))
```

The part in italics constitutes the information which is accessible in the concept past-participle-prefix through inheritance. It must not be specified explicitly. We only provide it here as an example of the effect of inheritance.

```
(DEFCONCEPT SUFFIX
     (A BOUND-MORPHEME
       (BOUNDARY (A MORPHEME-BOUNDARY)) ; This is a default,
                                        ; not always true.
       ((APPEND (?CONTEXT)
         (A STRING
           {Concatenate the following strings:
              (>> REFERENT OF context)
              (>> REFERENT LEXICAL-REPRESENTATION BOUNDARY)
              (>> REFERENT LEXICAL-REPRESENTATION)}
            ))))

(DEFCONCEPT PAST-SINGULAR-SUFFIX
     (A SUFFIX
       (BOUNDARY (A WORD-BOUNDARY))      ; Overrules the default.
       (LEXICAL-REPRESENTATION [STRING "Də"])))

(DEFCONCEPT PAST-PARTICIPLE-SUFFIX
     (A SUFFIX
       (LEXICAL-REPRESENTATION [STRING "D"])))
```

Figure 2. Morphological concepts in KRS (continued).

```
(>> REFERENT
  (APPEND
    (>> (APPEND [STRING "werk"])
        OF PAST-PARTICIPLE-SUFFIX))
  OF PAST-PARTICIPLE-PREFIX))
```

This will return *gə#werk+D.*

First, the embedded definite description is computed: *past-participle-suffix* inherits the *append* subject from type *suffix*. The application of this procedure with as argument a string with referent *"werk"* results in the concatenation of this string, the lexical representation of the boundary of the suffix (which is found through the lexical representation subject of *morpheme-boundary* to be "+"), and the lexical representation of the suffix itself.

The result of the concatenation is a string with referent (*werk+D*). It is used as an argument to the append method (which is inherited from *prefix*) of *past-participle-prefix*. Again, a concatenation is effected, this time of the lexical-representation of the prefix (which is *gə*), the boundary (a *word-boundary*, and therefore realised #), and the context, which is the newly created string with referent *werk+D*. The final result is a string with referent *ge#werk+D*. The example shows that fairly complicated sequences of actions and decisions can be described very simply if the necessary knowledge is organised object-orientedly.

Additional concepts have been defined describing *present-singular-suffix*, *plural-suffix*, and *present-participle-suffix*. The concept-hierarchy described so far is graphically represented in Figure 3. The inventory of types is by no means complete; for the morphology of nouns and adjectives, additional types should be created (for example a type representing *distance affixes* such as *ge...s* in *gezusters*, sisters).

At this point we would like to enlarge upon the lexical representation we use. As can be noticed from the examples already given, this representation incorporates graphemes, phonemes, morphophonemes (like *D*, for *t or d* in the computation of the past tenses of regular verbs) and a number of morphological markers (mainly boundary symbols). Both spelling and phonological representations can be derived from it by means of filters. Its main inspiration is of course the traditional *underlying form* in generative phonology, but it may have some psychological validity as well. An interesting syllable monitoring experiment by Taft and Hambly (1985) yields evidence for an abstract representation level incorporating morphemic structure and influenced by orthography. Both pronunciation and orthography could be generated from this level (see also Jakimik et al., 1985, and Aronoff, 1978 for linguistic arguments). In our own approach, the particular form of the lexical representation was a natural result of working with spelling as point of departure and of needing phonological information to compute word forms, and boundary information to compute both

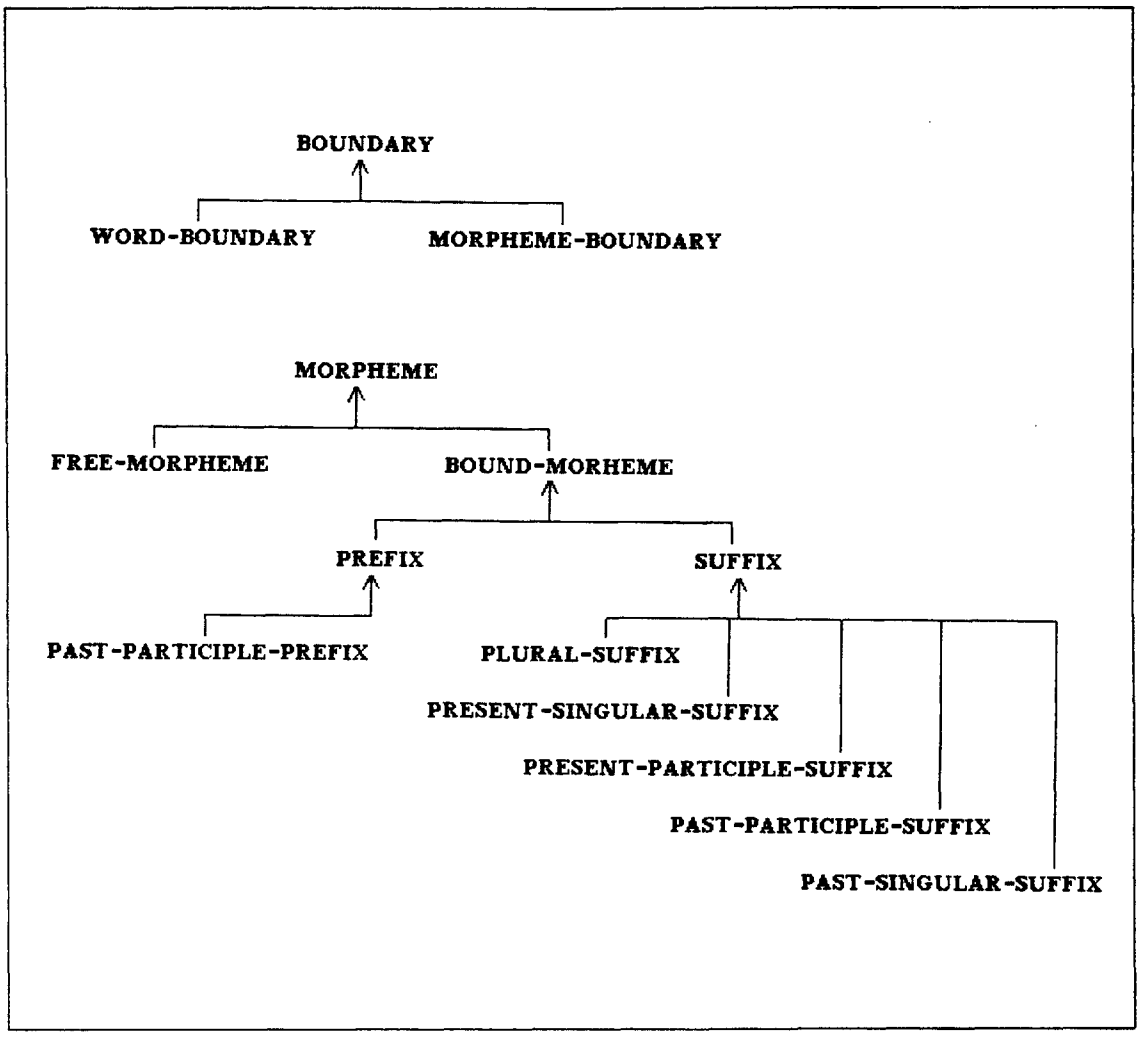


Figure 3. Part of the concept hierarchy for morphology.

spelling and phoneme representations.

3.1.2 Regular Inflection

A concept *regular-verb-lexeme*, specialisation of *verb-paradigm*, represents the paradigm of verbs with a regular inflection. When the inflection of a verb unknown to the system is asked, it is made a specialisation of this type as a first (default) hypothesis. The regular-verb-lexeme concept contains a number of subjects referring to procedures to compute the various verb forms belonging to its paradigm (Figure 4). The *root* of a verb-lexeme is an abstract entity (never realised in spelling or speech), but is necessary in the computation of actual verb forms. The *paradigm* subject of a regular-verb-lexeme lists all inflected forms, computed on the basis of the

```
(DEFCONCEPT REGULAR-VERB-LEXEME
   (A VERB-PARADIGM
     (CITATION-FORM (A STRING))
     (PARADIGM
       (A CONCEPT-LIST
          {List the following verb forms:
            (>> PRESENT-SINGULAR-ONE)
            (>> PRESENT-SINGULAR-TWO)
            (>> PRESENT-SINGULAR-THREE)
            ...
            (>> PAST-PARTICIPLE)}
       ))
     (ROOT
       (A MORPHEME
         (LEXICAL-REPRESENTATION
         {If
             (>> REFERENT CITATION-FORM)
             ends in a consonant + "iën", Then chop off last letter,
             Else: chop off two last letters}
          )))
     (PRESENT-SINGULAR-ONE
        (A VERB-FORM
          (LEXICAL-REPRESENTATION
           {Apply present-singular-one-rule to the citation form}
     (PRESENT-SINGULAR-TWO ...
     ...
     ))
```

Figure 4. Part of the concept *regular-verb-lexeme.*

root or other inflected forms.[3] In each subject referring to a morphological process, a concept is created which is a specialisation of the concept *verb-form.* In this object, slots are defined relating to appearance (spelling, lexical-representation), origin (its lexeme) and morphological features (finiteness, tense, number, person) of the verb form (Figure 5). The latter features are concepts in their own right.

The subjects of verb-forms are filled when their lexical representation is computed. A *linear* version of these created word forms can be stored in the lexical database as a dictionary entry (see sections 3.3 and 8.2). The lexical-representation subject of a verb-form is computed by applying a *morphological rule* to the root of the regular-verb-lexeme. An example of the internal structure of the subject present-singular-three of regular-verb-lexeme and a description of the relevant morphological rule are given in Figure 6. This figure also lists the description of the concepts *linguistic-rule* and *morphological-rule* of which present-singular-three-rule is a

[3] A KRS-expression like (>> PRESENT-SINGULAR-TWO) when used within the scope of a concept x is equivalent to (>> PRESENT-SINGULAR-TWO OF X). Within the subjects of a particular concept, the concept itself can be referred to by using the expression (>>). This is analogous to the *self* symbol in other object-oriented languages.

```
(DEFCONCEPT WORD-FORM
    (A FREE-MORPHEME))

(DEFCONCEPT VERB-FORM
    (A WORD-FORM
        (SPELLING ... )
        (LEXEME ... )       ; Pointer to the citation form
                            ; of which it is derived.
        (FINITENESS ... )   ; One of finite, infinite.
        (TENSE ... )        ; One of past, present.
        (NUMBER ... )       ; One of singular, plural.
        (PERSON ... )       ; One of first, second, third.
```

Figure 5. Part of the *verb-form* concept.

specialisation. The concepts shown here are simplifications of the ones actually used. We will come back to the rule concepts in Chapter 4.

In the same vein, subjects and rules were written to compute other inflected forms of verbs (present-singular-1, present-singular-2, present-plural, present-participle, past-singular, past-plural and past-participle). It can be inferred from these procedures that forms are computed by passing the root of the verb or other forms to the application subject of the morphological rule in question. This rule then makes use of the boundary and affix concepts described earlier to compute the lexical-representation.

The caching and lazy evaluation present in KRS are useful here because procedures sometimes make use of the result of other procedures. *Caching* means that fillers for subjects are computed only once, after which the result is stored. The consistency maintenance system built in provides the automatic un-doing of these stored results when changes which have an effect on them are made. Without these mechanisms, some procedures would have to be computed several times. E.g., the past plural procedure gives rise to a cascade of procedure applications: past-plural uses past-singular, past-singular uses root. Furthermore, some verb forms, which are not described here because they are identical to other forms or because their computation is trivial (imperative and conjunctive forms are a case in point) can be defined in terms of the procedures described above, without need for re-computation. *Lazy evaluation* means that the concept filling a subject is only computed when it is asked for. That way, many concepts can be defined without computational overhead, since computation of the referent of a concept is postponed until it is needed.

```
(DEFCONCEPT REGULAR-VERB-LEXEME
   ...
   (PRESENT-SINGULAR-THREE
     (A VERB-FORM
        (FINITENESS FINITE)
        (TENSE PRESENT)
        (NUMBER SINGULAR)
        (PERSON THIRD)
        (LEXICAL-REPRESENTATION
          (A STRING
            (DEFINITION
              [FORM
                (>> REFERENT APPLICATION OF
                     (A PRESENT-SINGULAR-THREE-RULE
                         (DOMAIN (>>)))))]))))   ; The rule is accessed
        ; with the particular regular-verb-lexeme as its domain.
       ...)))

(DEFCONCEPT LINGUISTIC-RULE
      (ACTIVE-P (A BOOLEAN))  ; With this predicate a rule can be
                              ; turned on and off.
      (CONDITIONS (A FORM))   ; If-part of the rule.
      (ACTIONS (A FORM))      ; Then-part of the rule.
      (APPLICATION (DEFINITION
           [FORM (IF (EVAL (>> REFERENT CONDITIONS))
                     (EVAL (>> REFERENT ACTIONS))
                     FALSE)])))

(DEFCONCEPT MORPHOLOGICAL-RULE
      (A LINGUISTIC-RULE
         (DOMAIN)             ; This domain is the context in which
                 ; conditions are checked and actions performed.
         (CONDITIONS TRUE)))  ; A morphological rule always applies
                 ; when it is used.

(DEFCONCEPT PRESENT-SINGULAR-THREE-RULE
      (A MORPHOLOGICAL-RULE
        (ACTIONS ; The root of the domain is appended to
                 ; the present-singular-suffix.
          [FORM (>> REFERENT APPEND
                    (>> LEXICAL-REPRESENTATION ROOT DOMAIN))
                OF PRESENT-SINGULAR-SUFFIX)])))
```

Figure 6. Apparatus to compute third person singular of verbs.

3.1.3 The Spelling Filter

So far, we have been computing *lexical representations* of word forms. As has already been explained, this level of description contains boundary symbols and morphophonemes, like *D,* which are not realised as such in spelling and sound. Spelling is derived from this lexical represention by means of a number of *spelling rules,* pronunciation by means of a number of phonological rules. Instead of including the necessary modifications in the different verb-form computation procedures or instead of introducing spelling subjects to the different affixes — thereby complicating them — we have adopted an approach in which a spelling filter, attached to the spelling

slot of the verb-form concept combines all spelling modifications. That way, more generalisations can be made.

Generalisation in an object-oriented system is represented by the attachment of subjects to concepts from which several other concepts inherit. Even more generalisations are possible if the system is extended with the inflections of adjectives and nouns as well. The (extended) spelling filter can then be moved up in the hierarchy from verb-form to word form or even free-morpheme.

The spelling filter consists of six spelling rules, specialisations of the type spelling-rule (Figure 7).

(1) *Consonant-degemination* is a rule which can be used to account for phenomena like the three following ones.

(i) Degemination of the final root consonant as in *legg*, *legg+t* etc. (realised as *leg*, *legt*).

(ii) Assimilation of suffix in past participle as in *ge#praat+t* and *ge#dood+d* (realised *gepraat*, *gedood*).

(iii) Assimilation of suffix in present singular as in *praat+t* (realised *praat*).

Note that we have to take into account the morphological boundaries (the rule must be blocked in cases like *praat#te*, past singular of *praten*, and realised *praatte*). More realistically, the scope of this rule is the syllable, which is not yet represented in the system (but will be in Chapter 4). Word-boundaries always coincide with syllable-boundaries while morpheme-boundaries are 'invisible' to syllabification rules. This explains the difference between 'praat+t' (degemination) and 'praat#te' (no degemination). In Chapter 4, phonological data and processes will be added to the spelling system, making reformulations of existing rules possible, without making them superfluous.

(2) *Zv-devoicing* as a spelling rule has a phonological correlate in *final devoicing*, a rule which states that all voiced obstruents are pronounced voiceless in syllable-final position. In spelling, this rule is restricted to *z and v* in root-final position. Examples are *blijv* (root of to stay) becoming *blijf*, and *huiz* (root of to house) becoming *huis*. There is one exception to this rule; the loan word *fez*.

(3) An example of *vowel-doubling* is the fact that *lat* (root of *laten*, to let) becomes *laat*. The conditions in the rule avoid the generation of the following ungrammatical verb forms: **aankondiig*, **duuw*, **waaai*, **houud* etc. Forms like *appreciëren* (to appreciate) which seem to be wrongly adapted by this rule at

```
(DEFCONCEPT SPELLING-RULE
     (A LINGUISTIC-RULE
        (DOMAIN)
        (CONDITIONS TRUE)))

(DEFCONCEPT CONSONANT-DEGEMINATION-RULE
     (A SPELLING-RULE
         (ACTIONS
         {If lexical representation contains a word-final
          geminate C_iC_i or C_i+C_i, then
          replace C_iC_i by C_i})))

(DEFCONCEPT ZV-DEVOICING-RULE
     (A SPELLING-RULE
         (ACTIONS
         {If lexical representation contains z or v followed by
          nothing or by a morpheme boundary not followed by a
          vowel, then replace z by s or v by f})))

(DEFCONCEPT VOWEL-DOUBLING-RULE
       (A SPELLING-RULE
          (ACTIONS
          {If a vowel V, equal to a, e, o or u is preceded by
           a segment which is not equal i, a, e, or o, and followed
           by a segment which is not equal to w or i, and the
           latter is followed either by nothing or by a morpheme
           boundary, which in its turn is not followed by a vowel,
           then replace V by VV in the lexical representation}

(DEFCONCEPT SPELLING-ASSIMILATION-RULE
       (A SPELLING-RULE
          (ACTIONS
          {If morphophoneme D is preceded either by t followed by
           a word boundary or by one of p, s, k, f, h followed by
           a morpheme boundary, then replace D by t. If D is
           preceded by t or d followed by a morpheme boundary, then
           replace D by "" (empty-string). Else, replace D by
           d.})))

(DEFCONCEPT SCHWA-REDUCTION-RULE
      (A SPELLING-RULE
          (ACTIONS
          {If a schwa is followed by a morpheme boundary followed
           by a schwa, then delete the second schwa})))

(DEFCONCEPT GE-DELETION-RULE
      (A SPELLING-RULE
          (ACTIONS
          {If gə# is followed by one of ont, her, vər,
           bə or gə, then delete gə#})))
```

Figure 7: Spelling Rules.

first sight (*apprecier* instead of *apprecieer*), are nevertheless correctly formed because the diaeresis is kept in the lexical representation of a word; it is represented as a special symbol which precedes the character to which it applies. The presence of this code 'prevents the preventing' of the application of this rule.

(4) *Spelling assimilation* is a rule which states that the voicedness of the first consonant of past-participle and past-singular suffixes (represented at the lexical level by "D") depends on the voicedness of the final segment of the morpheme to which they are appended. It is clearly a sandhi-process, but a marked one, since the direction of assimilation is opposed to the normal (phonological) case. It is therefore treated as a spelling rule.

(5) Another rule, *ge-deletion* states that the realisation in spelling of the past-participle prefix is either *ge* or an empty string, depending on the presence of certain affixes in the morpheme to which it is appended. E.g. compare *ge#werk+t* (worked) of *werken* to *be#werk+t* (adapted) of *bewerken*. In the latter case, the prefix is not realised. A category of problematic verbs exists (e.g. *vernissen*, to varnish, past participle *vernist* and in some dialects *gevernist*). We evaded the problem by storing these verbs without an internal word boundary between the prefix and the root.

(6) Finally, *schwa-reduction* removes a suffix-initial schwa (*ə*) when it is preceded by one. E.g. *maak#tə+ən* becomes *maak#tə+n* (they made),

Apart from these spelling rules, the spelling slot in *verb-form* contains procedures which remove boundary symbols, insert a dieresis if necessary (to separate vowels if mispronunciation is likely, see Chapter 4), and transform remaining lexical representation symbols into their spelling correlates. For example, *ə* becomes *e* etc.

Notice that the order in which spelling rules are applied is important; vowel doubling can occur after zv-devoicing, as in

blaz -(zv-devoicing)-> *blas* -(vowel-doubling)-> *blaas*,

but not after consonant degemination:

legg -(degemination)-> *leg* -(vowel-doubling)-> **leeg*

Generally, the rules are disjunctively ordered[4]. They interact only in two places: spelling-assimilation should precede zv-devoicing, and vowel doubling may apply after consonant degemination. In our program, the ordering of the rules guarantees the correct rule interactions. Provision must also be made for a recursive application of consonant degemination; e.g. in *putt+t* (he draws), realised as *put*.

[4] This means that they are mutually independent and can be applied in parallel.

3.1.4 Interaction with Phonology and Error Diagnosis

We have already mentioned syllabification as a phonological process which has an effect on word formation (the syllable is a domain in spelling rules). In some other cases, too, the processes described so far cannot operate properly on the basis of the spelling image of the morphemes alone. This is due to an ambiguity in Dutch spelling, where grapheme <e> does not distinguish between phonemes /ɛ/, /e/ and /ə/ (unstressable vowel). This interferes with verb form computation in two spelling rules: *ge-deletion* and *vowel doubling*.

Some of the prefixes blocking the realisation of the past-participle prefix (notably *ge, ver* and *be*) cannot be reliably identified on the basis of their spelling. E.g. *verven* /vɛrvən/ (to paint) has past participle *gə#vɛrf+d* while *verbeteren* /vərbetərən/ (to correct) has *ver#beter+d*. In the latter case the prefix is not realised.

The vowel doubling rule is blocked when the vowel to be doubled is a /ə/. E.g. *vervelen* /vərvelən/ (to bore), *ik verveel* (I bore) versus *wervelen* /wɛrvələn/ (to whirl), *ik wervel* (I whirl). In some cases, the difference can only be made on semantic grounds, e.g. *bedelen* /bədelən/ (to endow) or /bedələn/ (to beg). But often only one phonological representation is possible for a particular spelling image.

The best solution is to include a phonological level and a level of morphological analysis (prefix stripping) in the model. To this end, we have to extend our morphological database with concepts for phonological objects and processes. Such a level will be described in the next chapter. In the absence of phonological information, heuristics could be used. E.g. consider the case of the disambiguation of grapheme <e>. In a word with three instances of <e> (e.g. *wervelen*, to whirl), 3^3 combinations of /e/, /ɛ/ and /ə/ are theoretically possible. The number of possibilities can be reduced to three by taking into account heuristics like the following: Dutch infinitives end in /ən/, Dutch morphemes must have at least one vowel not equal to ə, when <e> is followed by two identical consonants it represents /ɛ/, etc.. In the case of three consecutive <e>s, the three remaining analyses are ɛ-ə-ə, ɛ-e-ə, and ə-e-ə. All possibilities are realised in existing Dutch words: *wervelen* (to whirl), *herlezen* (reread) and *vervelen* (to bore), respectively. The heuristics we developed can disambiguate all words with two consecutive syllables containing grapheme <e>. The system could also ask the user for information each time it encounters a choice which it cannot make with the available information. In that case heuristics are still useful to constrain the number of possibilities among which the user has to choose.

We adopted a strategy in which 'educated guesses' are used to make choices in the absence of the necessary information. The user, monitoring the output of the program, indicates erroneous forms or parts of forms (this can be done quickly and easily with a mouse and menu system), and on the basis of this information, the system tries to diagnose its error and to find a correction. To achieve this, a diagnosis procedure is attached to the regular-verb-lexeme object, which accepts as input information about how many and which forms were indicated as erroneous, and delivers as its result a new paradigm of computed forms. As a side-effect, this procedure can delete false instances of the concept verb-form, and recompute them. Only anticipated errors can be handled this way. If no errors are detected in a computed paradigm, the computed verb forms are added to the lexical database.

The same auto-correction procedure is used to recover from errors due to an erroneous categorisation of input verbs (a verb with irregular inflected forms is erroneously made a specialisation of the regular-verb-lexeme type). In this case, the diagnosis procedure specialises the verb-lexeme as one of the (semi-)irregular verb categories discussed in the next section, and transfers control to this lexeme, which may have a diagnosis method itself.

3.1.5 Irregular Inflection

The inflection of Dutch 'strong' verbs involves a vowel change which is a relic of an Ablaut process which has become unproductive. The different classes of irregular verbs must be learned, and a possible reason for their survival is their high frequency of use.

A specialisation hierarchy of irregular-verb concepts is used to describe the inflection of these irregular classes. Specialisations of these types inherit regular procedures from the regular verb lexeme type, and irregular procedures are delegated to other categories of (semi-)irregular types or to concepts which represent exception mechanisms (we will call the latter *mixins*). This process of multiple inheritance (an object inherits from two or more types) allows an easy definition of exception categories. Figure 8 lists the mixins we use for Dutch irregular verbs. Inheritance by delegation is only one possibility to implement multiple inheritance in KRS, but in this case it is the most natural way. Irregular verbs mostly are only irregular in a few forms (mainly past forms). Only for these forms, irregular processes must be considered. This can be done best by directing inheritance explicitly to some type or

mixin for these forms.

```
(DEFCONCEPT IRREGULAR-VERB-MIXIN)

(DEFCONCEPT VOWEL-CHANGE-MIXIN
   (AN IRREGULAR-VERB-MIXIN
       (PAST-ROOT
        {Change vowel in (>> ROOT)})
       (PAST-PARTICIPLE-ROOT
        {Change vowel in (>> ROOT)})
       (PAST-SINGULAR
         (A VERB-FORM
            ...
            (LEXICAL-REPRESENTATION
              {Compute past-singular using past-root}
       (PAST-PARTICIPLE
         (A VERB-FORM
            ...
            (LEXICAL-REPRESENTATION
             {Compute past-participle using past-participle
              root}))))))


(DEFCONCEPT EN-PAST-PARTICIPLE-MIXIN
   (AN IRREGULAR-VERB-MIXIN
       (PAST-PARTICIPLE
         (A VERB-FORM
           ...
           (LEXICAL-REPRESENTATION
            {Compute past-participle using
             en-past-participle-rule}))))))
```

Figure 8: Concepts for irregular verb mixins.

Sometimes, the vowel change in the *vowel-change* mixin can be predicted with certainty. E.g. an irregular verb with root vowel *ui* has a past participle and past root with vowel *o*. Sometimes, there are two possibilities. In that case, the most probable solution is chosen as a first try. With the present diagnosis system, a maximum of two tries is needed to predict the vowel change of all existing strong verbs. The consonant frame of the root, too, can be of help in predicting the vowel change. As irregular verbs constitute a closed class, it is of course possible to store them all, but such a solution would do injustice to the regularity which clearly exists in the Dutch irregular verb system.

The *en-past-participle* mixin uses the past-participle root if it is defined (in that case, the concept inherits from vowel-change mixin), otherwise, the root of the *regular-verb-lexeme* type. The spelling rules and concept hierarchy used in the computation of regular verbs are also applicable here. Only one suffix (*en*) had to be added (Figure 9).

```
(DEFCONCEPT EN-PAST-PARTICIPLE-SUFFIX
    (A SUFFIX
        (LEXICAL-REPRESENTATION
            [STRING "ən"])))
```

Figure 9. The concept *en-past-participle-suffix*

Irregular verb categories can be defined in terms of the *regular-verb-lexeme* type and the *irregular-verb-mixins* (Figure 10).

```
(DEFCONCEPT SEMI-IRREGULAR-1-VERB-LEXEME
    (A REGULAR-VERB-LEXEME
      ((PAST-PARTICIPLE
        (A DELEGATING-SUBJECT
           (DELEGATE-TO (AN EN-PAST-PARTICIPLE-MIXIN))))))
```

For example, *bakken* (to bake), *bakte*, *gebakken*.

```
(DEFCONCEPT SEMI-IRREGULAR-2-VERB-LEXEME
    (A REGULAR-VERB-LEXEME
      ((PAST-PARTICIPLE
        (A DELEGATING-SUBJECT
           (DELEGATE-TO (A VOWEL-CHANGE-MIXIN))))))
```

For example, *wreken* (to revenge), *wreekte*, *gewroken*.

```
(DEFCONCEPT SEMI-IRREGULAR-3-VERB-LEXEME
    (A REGULAR-VERB-LEXEME
      ((PAST-SINGULAR
        (A DELEGATING-SUBJECT
           (DELEGATE-TO (A VOWEL-CHANGE-MIXIN))))))
```

For example, *vragen* (to ask), *vroeg*, *gevraagd*.

```
(DEFCONCEPT IRREGULAR-1-VERB-LEXEME
    (A SEMI-REGULAR-1-VERB-LEXEME
      ((PAST-SINGULAR
         (A DELEGATING-SUBJECT
           (DELEGATE-TO
             (A SEMI-IRREGULAR-3-VERB-LEXEME))))))
```

For example, *lopen* (to run), *liep*, *gelopen*.

```
(DEFCONCEPT IRREGULAR-2-VERB-LEXEME
    (A SEMI-IRREGULAR-2-VERB-LEXEME
      ((PAST-SINGULAR
         (A DELEGATING-SUBJECT
            (DELEGATE-TO
              (A SEMI-IRREGULAR-3-VERB-LEXEME))))))
```

For example, *zwijgen* (to keep silent), *zweeg*, *gezwegen*.

Figure 10. Concepts for irregular verb categories.

In principle, every single irregular verb can be defined in terms of these and similar exception mechanisms. However, for classes with only one or a few items, the exceptional inflections were defined explicitly for reasons of efficiency. This is the case with suppletive *zijn* (to be), auxiliaries, and verbs with a consonant change (e.g. *kopen*, to buy, past singular *kocht*). In appendix A.3.1, the complete concept hierarchy used in the computation of verb forms is shown, and derivations of regular as well as irregular verbs are given.

Figure 11 gives a survey of the complete synthesis part of the program. Unknown citation forms are made an inheritor of one of the paradigms (lexemes) of which the attached morphological processes are used to compute the lexical representation of inflected forms (by means of the affixes and the stems). This lexical representation is transformed into spellings or pronunciations by means of a spelling and a pronunciation filter, respectively. The pronunciation filter will be described in Chapter 4.

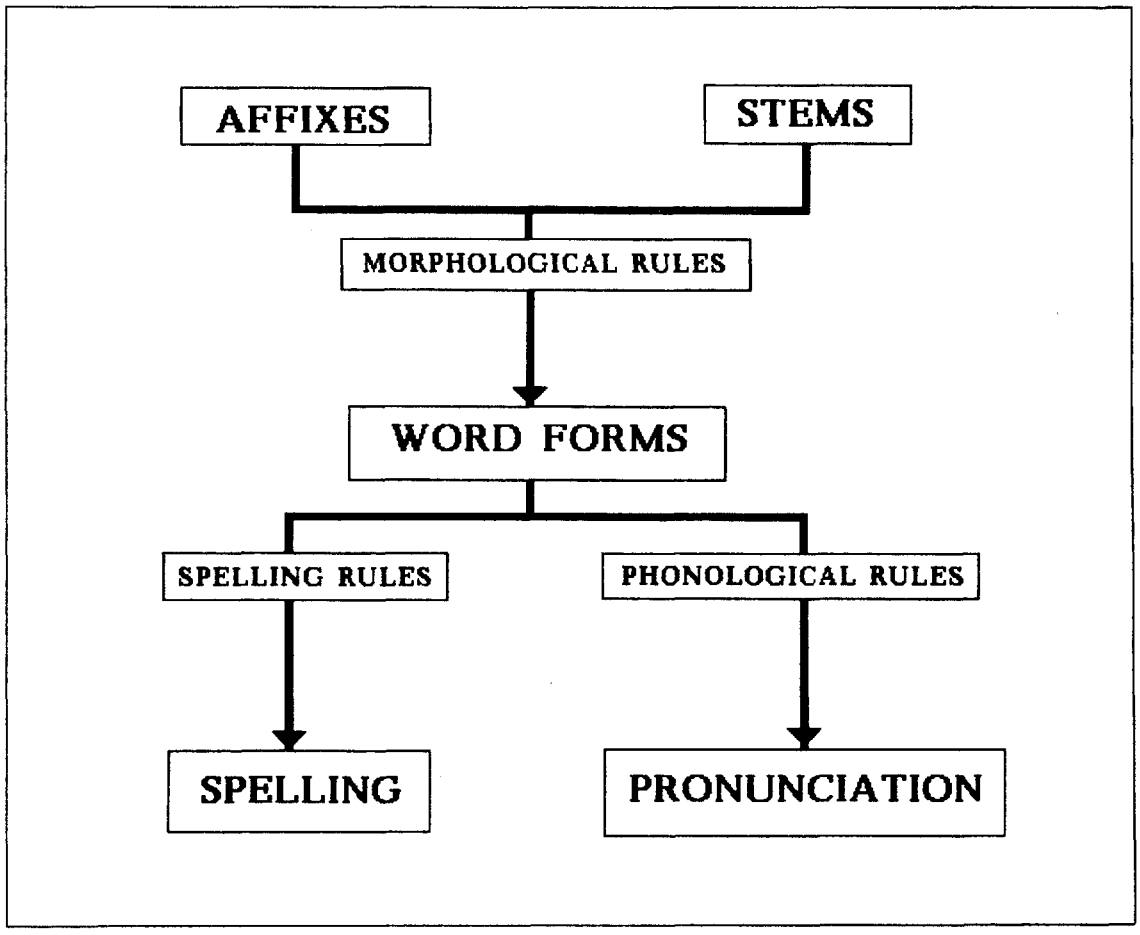


Figure 11. Overview of the synthesis algorithm.

3.2 Morphological Analysis

The task of a morphological analysis algorithm is to determine

(1) whether a given string of orthographic or phonological symbols constitutes a word of the language,

(2) if this is the case,

(2a) to represent its internal (lexical) structure and

(2b) to access its linguistic properties.

Such an algorithm presupposes the existence of a *lexical database* and a *morphological grammar* as data, and *segmentation* and *parsing* procedures as processes. The presence of a grammar makes it feasible for a program to recognise not only existing words, but also *possible words*. These are words which do not exist (yet), but may one day be created as a result of rule-governed creativity. Some Dutch examples are *louchiteit* and *voordeurdeler*. These words did not exist until recently, but are perfectly acceptable for any speaker of Dutch.

As was noted before, morphological analysis is undetermined due to ambiguity. E.g. a spelling string *lopen* would have to be analysed as in Figure 12.

```
lop+en (noun plural) of LOOP                          (barrels)
lop+en (verb nominalised) of LOOP                     (the running)
lop+en (verb finite present plural person-1) of LOOP  (we run)
lop+en (verb finite present plural person-2) of LOOP  (you run)
lop+en (verb finite present plural person-3) of LOOP  (they run)
lop+en (verb infinitive) of LOOP                      (to run)
```

Figure 12. Six Analyses of *lopen*.

We will not be concerned with how the linguistic (and in some cases extra-linguistic) context determines the *appropriate* analysis; our algorithm will have to provide all reasonable analyses.

Applications of morphological analysis include hyphenation and spelling error correction (Chapters 5 and 6), integration into the interpretation parts of larger natural language processing systems (machine translation and dialogue systems), and systems for Computer Assisted Instruction (Chapter 7).

3.2.1 The Storage versus Processing Controversy

In the design of an analysis system, decisions must be taken about the balance between the amount of information included in the dictionary (storage) and the scope of the analysis algorithm (computation). The complexity of an analysis algorithm is directly proportional to the size and nature of the dictionary.

Different types of dictionary can be distinguished depending on the kind of entries which are stored: *morpheme* dictionaries, *citation form* dictionaries and *word form* dictionaries. Most existing dictionaries (printed as well as computer-readable) only contain the *citation form* of a particular word (infinitive for verbs, singular for nouns etc.) and sometimes also some irregular forms (e.g. the past tense *ran* of *to run* would be entered as a separate entry because it is inflected irregularly)[5]. *Morpheme* dictionaries only list the morphemes (base forms and affixes) of the language.

If we want to use these relatively small dictionaries in morphological analysis, we have to build a complicated morphological analysis program, since we should be able to find the base form of derived forms which are often different from the form listed in the dictionary. E.g. we have to recognise *huizen* as the plural of *huis* (house), *vertrouwd* as the past participle of *vertrouwen* (to trust), *koninkje* as the diminutive form of *koning* (king), *huizenkoninkje* as a compound based on *huis* and *koning*, *autogereden* as the past participle of the compound verb *autorijden* (to drive), etc. If, on the other hand, we have a *word form* dictionary in which all derived forms of a citation form (irregular and regular) are listed as entries — i.e. if the whole paradigm is entered; *tafels, tafeltje* and *tafeltjes* of *tafel* (table); *werk, werkt, werkte, werkend, gewerkt* ... of *werken* (to work) etc. — the morphological analysis program can be much simpler; it only has to look for compounds and is relieved of the burden of having to interpret conjugations and declensions.

We have adopted the latter strategy because (1) this form of analysis is more efficient in processing time, and (2), for Dutch it is feasible to store all derived forms. We will discuss both claims.

---

[5] Makers of dictionaries do not always seem to know exactly which form is exceptional and which is not. E.g. the first person singular of *komen* (to come) is *ik kom* and not the regular form *ik koom*, but it is never listed in a dictionary. On the other hand, the perfectly predictable allomorphs of the Dutch diminutive suffix *-tje* (*-pje, -kje, -etje, -je*) are often listed as exceptions.

(1) The overall speed of program execution depends on a large number of factors. If we disregard those which have to do with installation dependent parameters (processor clock frequency, complexity of machine instruction set etc.), two main components remain: the number of *dictionary accesses* (one access is the mean time it takes to determine whether a string of letters or phonemes is present in the dictionary), and the *algorithm processing time* (the time it takes for the segmentation and parsing procedures to analyse the input). In general, an analyser with a small dictionary will consume more algorithm processing time and need more accesses, but access time will be shorter (access time is a monotonously increasing function of the size of the dictionary). Only if the difference in access time for word form and morpheme (or citation form) dictionaries is very high (which is not the case for Dutch with current storage technology), the morpheme dictionary solution would be faster than the word form dictionary solution. The dictionary we use, as well as various possible storage and search techniques, are fully discussed in section 3.3.

(2) Provided an algorithm exists which generates (semi-)automatically the derived forms of the citation forms, a word form lexical database can be easily constructed and updated. Part of such a program was described in section 3.1. Furthermore, technologically speaking, the storage of large amounts of word forms causes no problem.

The amount of processing necessary to analyse a particular word form depends on the place this word form takes in the *regularity continuum*. On one extreme of this continuum we find the *trivial* cases, on the other extreme the completely *exceptional cases*, and in between forms of varying levels of (ir)regularity. A trivial form is defined as being analysable using a simple, exceptionless rule. E.g., a diminutive plural is always formed by adding 's' to the diminutive singular form (*hond#je+s*, little dogs). Completely exceptional forms on the other hand cannot be analysed using rules. E.g., the citation form of suppletive verb *waren* (were) must be looked up, and cannot be computed (the form is *zijn*, to be). In most approaches to morphological analysis the boundary between exceptional and regular is put arbitrarily somewhere in the middle of the regularity continuum. Forms which are regular according to this ad hoc boundary are computed, and the other forms are stored. In our own approach, only the trivial forms are computed (using the exceptionless rule criterion) and all other forms are stored.

Summing up, the specific balance between processing and storage chosen in a particular system, depends in the first place on the available storage resources and on the application at hand. For our own purposes, an optimal balance for morphological analysis between processing and storage is achieved by storing the irregular and regular forms and computing only the trivial ones.

3.2.2 The Algorithm

Figure 13 gives an overview of the data, processes and representations in our analysis system. The remainder of this section contains a detailed description of these components.

Our implementation so far works with spelling input exclusively, but the same algorithm can be applied to phoneme input if a phonological dictionary is available. Such a dictionary can be created using the phonemisation algorithm described in Chapter 4. The spelling input is first transformed into a *normal form* (a form without special characters, uppercase letters, numerals etc.)[6], and is made a specialisation of a concept *possible-word-form*, which has a *segmentations* subject.

Segmentation. Attached to the segmentations subject is a procedure which finds possible ways in which the input string can be partitioned into dictionary entries. First it is checked whether the complete string is present in the dictionary; if so, the analysis algorithm returns the associated analyses. Then, increasingly longer left substrings of the input string are taken in an iterative way, and the remaining right part is looked up in the dictionary. If the latter is present, the whole procedure is applied recursively to the (left) remainder of the string. The procedure may stop with the first solution ('longest-only') or continue looking for other analyses with smaller dictionary items ('all-possibilities'). Both options are implemented in the present system, but the longest-only solution is sufficient in most applications. It is also more efficient in processing time. When the 'grain size' of segmentation is small, more possible segmentations will have to be accepted or rejected by the parser; this leads to considerably longer algorithm processing times. E.g. in the famous *kwartslagen* example, not only *kwart+slagen* and *kwarts+lagen* would have to be considered, but also *kwart+sla+gen*, *kwarts+la+gen*, *kwart+slag+en* and *kwarts+lag+en*. The two last

[6] See Chapter 6 (spelling error detection) and 8 (lexical analyser) for a discussion of this step, which is not relevant in this context.

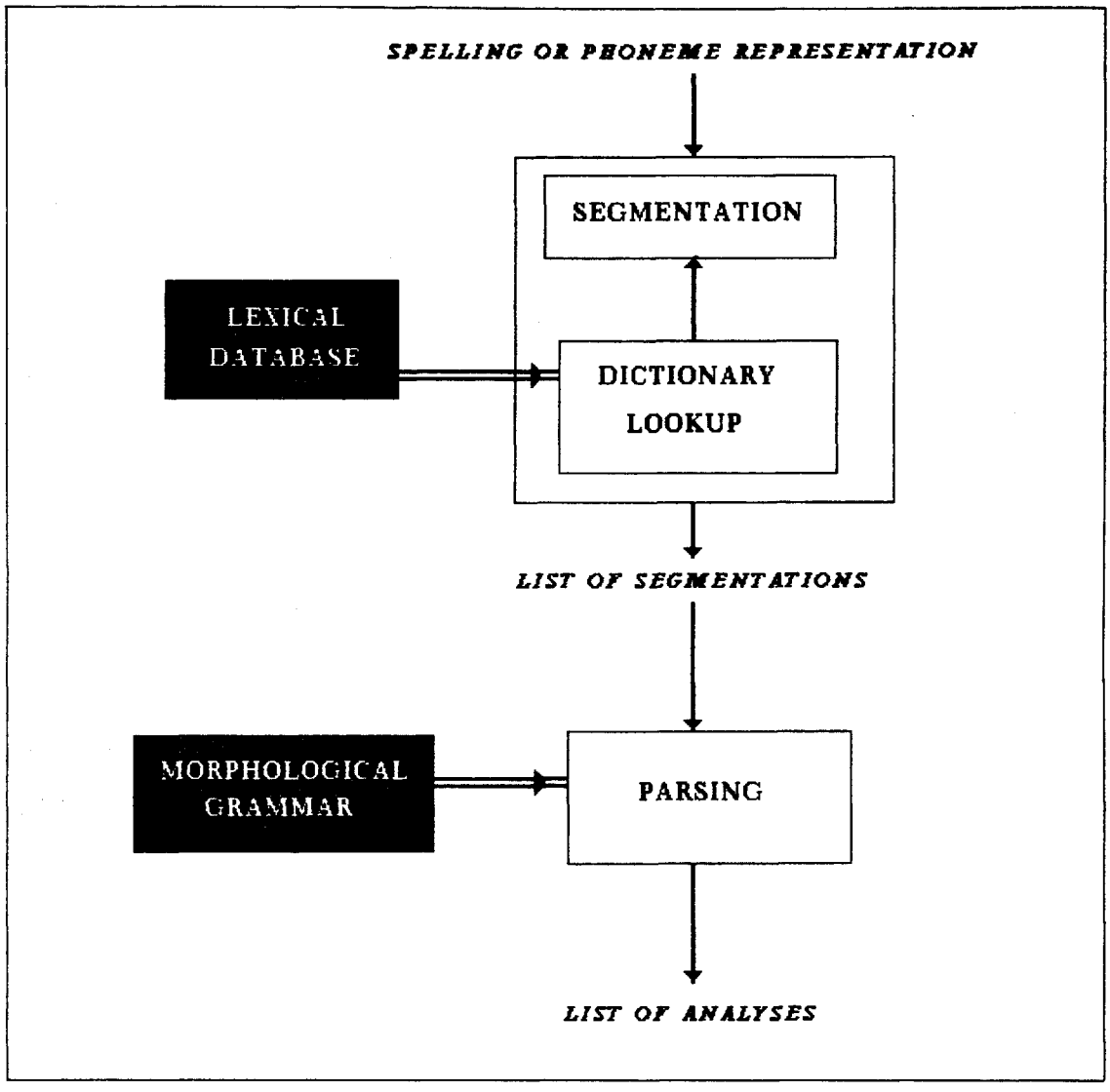


Figure 13. Data, representations and processes in a morphological analysis algorithm. White boxes are processes, black boxes are data. The double arrows indicate which data are used by which processes. The single arrows indicate the input and output relations between representations and processes.

segmentations would be rejected by the parser as the conjunction *en* (and) cannot feature in a compound. Related to this problem is an increased risk at the occurrence of nonsense-analysis results (errors by over-acceptance). E.g. *liepen* (ran) would be analysed as *li#epen* (epics about the Chinese measure *li*), *laster* (slander) as *la#ster* (drawer star), *kwartel* as *kwart#el* etc. The latter analyses are not wrong (they are allowed by the morphological grammar, and may possibly occur), but they are unpractical as they will place a burden on the disambiguation part of any program in which the analysis module is to function. A longest-only approach prevents this.

Furthermore, people producing them in spelling would be inclined to write a hyphen between the two parts of the compound. An additional reason to prefer a longest-only solution is the fact that we are working with a word form dictionary, in which boundary information is present. I.e. morphological structure can be *retrieved* for a large part, making further analysis superfluous.

There is no theoretical reason to prefer a segmentation using the longest right part to one using the longest left part. Both could be defended; the former by referring to the *head* role played by the second part of a compound, the latter by pointing at the 'psychological relevance' of left to right processing. We are currently using a longest right part approach, but the program can be easily adapted to a longest left approach.[7] When used in a longest-only mode, our algorithm does not immediately stop, but keeps on iterating two more characters. This provision was built in in order to be able to find ambiguity of the kind exhibited by *kwartslagen*. A longest-only, longest right part approach stops with *kwart#slagen*, but with one more iteration, *kwarts#lagen* is found as well. We think two iterations is enough to find the large majority of ambiguous cases with a minimal number of dictionary accesses. We have already mentioned that the program can also be run in a more expensive mode in which all possible combinations are found.

Dictionary Lookup. In order to check whether a substring of the input string constitutes a dictionary entry, a procedure *word-form-p* — a subject of the concept *string*, is used. If the word is found, this procedure returns categorial information relevant for the parser as well as the lexical representation of the word (possibly with information about internal word boundaries); a 'False' message is returned if the word is not found. Some global conditions on strings were devised to constrain the number of dictionary accesses necessary (thereby making the system more efficient).

(i) Strings with a length less than the shortest dictionary entry, or longer than the longest dictionary item are not considered for dictionary lookup (Cp. Brandt Corstius, 1978).

(ii) Strings which do not conform to the morpheme structure conditions of Dutch are not looked up. To achieve this, the initial and final consonant clusters of a string are checked before looking it up in the dictionary. If the string can be

[7] It would be interesting to investigate if there are any empirical differences when using a longest-only analysis between longest right and longest left. The difference could be measured in number of errors, number of dictionary accesses needed, etc.

rejected on the basis of these phonotactic restrictions, processing time can be saved This is of course only true if the string-manipulation and searching necessary for checking the restrictions demands less processing time than a single dictionary lookup. In our particular case, the time saved is considerable. More information about the use of phonotactic restrictions is given in Chapter 4 and 6.

(iii) All strings looked up get a property *already-analysed*, which can be Nil (if the string is not looked up yet), a list with dictionary information (if the string was already looked up, and found to be a word), or - (if the string was already looked up, but was not found). Thus, the result of the previous lookup is immediately available, which again constitutes a considerable gain in processing time. During segmentation, the same substrings are often looked up more than once. Again, a caveat is in its place here: in small systems, internal storage may be too small to follow this approach which requires additional memory to store the strings.

The lookup procedure also includes some spelling modifications. Sometimes, a linking grapheme emerges between the parts of a compound. E.g. boerekool, hemelsblauw, eierkoek etc. The occurrence of linking graphemes is largely unpredictable; no hard and fast rules exist (Van den Toorn, 1981a, 1981b, 1982a, 1982b). This is reflected in spelling behaviour: a lot of confusion exists about this aspect of Dutch spelling. We looked for a practical solution to this problem. The analysis system accepts all linking graphemes which cannot be ruled out on phonotactic grounds (e.g. *kaassplank can be ruled out on the basis of the double s). The 'trick' we use is the following. If a string, representing the left part of a possible compound and ending in s, is not found in the dictionary, it is looked up again without the s. If a string ending in e is not found, it is looked up with an n attached to it. Although it is a heuristic, it works surprisingly well. The problem of the er linking grapheme was solved by listing morphemes like *kinder* in the dictionary (they form an exceptional case and a small closed class with *kalver, blader, hoender, volker, kleder, lammer, runder* and *eier*).

Parsing. The result of segmentation and dictionary lookup is a set of segmentations, associated with the input form, and a number of strings (dictionary items with some categorial information and their lexical representation attached to them). Appendix A.3.2 lists some sample traces of the application of the segmentation and dictionary-lookup procedures. This environment will be used by the parsing subject associated

with the concept *possible-word-form* to compute the subset of segmentations which are allowed by the compound grammar of Dutch. The grammar consists of a number of restrictions on co-occurrence. E.g. Figure 14 lists the rule which restricts the behaviour of compounds with a noun as their second part.[8]

```
RULE  Noun =  X + Noun

      If      X = Noun
      Then    X = one of Singular Noun, Plural Noun,
                  Diminutive Plural Noun

      If      X = Adjective
      Then    X = one of Normal Form Adjective, Inflected Adjective

      If      X = Verb
      Then    X = Present Singular First Verb
```

Figure 14: Rule which restricts the formation of compound nouns.

The rule prevents the acceptance of ungrammatical constructions like **meisjegek, *hogerschool, *zwevenvliegtuig* etc. At the same time, if a combination is accepted, the rule assigns a lexical representation to it. The compound inherits all linguistic features from its second part (the head). Similar rules exist for possible combinations with adjectives and verbs. The parser works from left to right. E.g. with a segmentation A + B + C, first the legality of A + B is checked, and then the legality of D + C (with D = A + B, if A + B is a legal combination).

Measuring Efficiency. Efficiency of a morphological analysis program is a function of the number of dictionary accesses needed to assign an interpretation to the input string. In this paragraph we will relate the number of necessary dictionary accesses for varying input lengths (when using our algorithm) to the theoretically necessary number of accesses in the worst case.

It can be proved that the theoretical number of dictionary accesses necessary in the general case (without the restrictions described earlier) is an exponential function of the length of the input. The relation is described in formula (1).

[8] We adopt the view that in a single composition, a maximum of two word forms is combined. Viewed from that perspective, *huisvuilvernietigingsfabriek* is the composition of *huisvuilvernietiging*, *'s'* and *fabriek*, *huisvuilvernietiging* the composition of *huisvuil* and *vernietiging*, and finally, *huisvuil* the composition of *huis* and *vuil*. The final structure using labeled bracketing would be $[[[\text{huis}_N + \text{vuil}_N]_N + \text{vernietiging}_N]_N + \text{fabriek}_N]_N$.

(1) $$f(n) = 2^{n-2}+2f(n-1) \ (n > 0 \text{ and } f(1) = 1)$$

This implies that processing time approaches ∞ in exponential time. The introduction of restrictions (i) and (iii) above reduces this to polynomial time. The relation is expressed in formula (2).

(2) $$f(n) = (n^2 - 5n + 10)0.5$$

In an empirical test (see below for details) of our algorithm, which incorporates as additional restrictions (ii) above and a longest-only segmentation strategy, the number of accesses was further reduced to a linear relation. This means that access time increases linearly with the length of the input (which is acceptable for practical applications). For the worst case (i.e. the segmentation routine finds no segmentation), this linear relation approaches f(n) = 0.5n (where n is the length of the input string, and f(n) averages access time). Figure 15 depicts the functions for the three cases dealt with. Notice that in our empirical test, the number of accesses is further diminished by the fact that substrings already analysed in earlier input strings are 'remembered' as well, which results in an increasingly better performance of the algorithm.

The introduction of a phonotactic check before dictionary access reduced the required number of accesses with 15% on the average (for words with length 2 to 20). For words with length 10 to 20, the average gain even equalled 25%. In systems where checking phonotactic restrictions is expensive, the restriction could be reserved to larger words only.

Affix stripping. For some applications (notably hyphenation, chapter 4, and morphological synthesis), an analysis system which can find internal word boundaries (boundaries with representation #) is needed. Although we presume a dictionary in which these boundaries are already present, we nevertheless have to provide for the possibility that such a dictionary is not available (for example to indicate internal word boundaries in existing dictionaries, or for new words). To strip affixes, a list of prefixes followed by an internal word boundary and a list of suffixes preceded by an internal word boundary, are merged with the dictionary, and additional rules are added to the compound grammar. The normal analysis procedures described in this section can then be applied. Notice that in this case, a number of parts of formally complex words must be added to the dictionary which are not morphemes in the sense that they are meaningful units. They nevertheless have the same status: e.g. *amen* in *be#amen* (to agree) and *velen* in *be#velen* (to order) are cases in point (see Booij, 1977 and 1981). A similar form in English could be *cran* in *cranberry* (Aronoff, 1976).

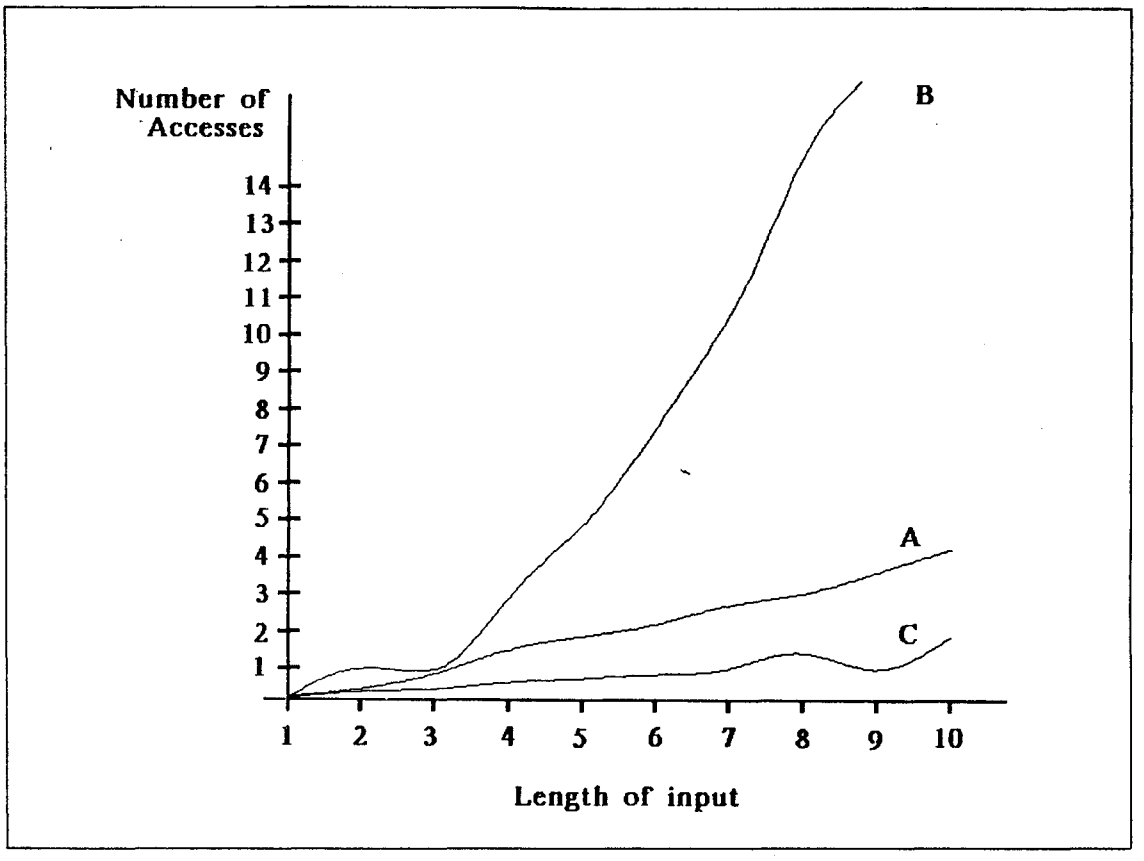


Figure 15. Relations between number of accesses and string length. Curve A corresponds to the *logarithm* of formula (1) in the text, curve B to formula (2), and curve C to the results of our empirical test.

Performance. We tested our parser on a number of compound words. Of an input text of 924 word tokens, each word token was analysed by the morphological parser. The speed of the program (measured in average number of dictionary accesses needed) has already been discussed earlier. The text contained 45 compounds (types, not tokens), 29 of which were correctly analysed (64%). For ten compounds (22%) no analysis was found because one of the parts was not present in the dictionary. We used only the Top-10,000 dictionary (described in section 3.3) as lexical database, without an additional user dictionary for special terminology. As the input contained a lot of specialised terminology (*checker, vocabularium, detectie, congruentie* etc.) the results should be interpreted with caution. Also due to the incompleteness of the dictionary, six compounds (14%) were analysed wrongly. E.g. as *massas* (masses) was not present in the dictionary, the word form was analysed *ma+s#sas*. Similarly, *tekstverwerker* (word processor) was analysed *tekst#ver#werker* (text far worker) due to the absence of *verwerker*, and *inputtekst* (input text) was analysed *in#put#tekst* (collect hole text) due to the absence of *input*.

Residual Problems. A more efficient architecture could probably be obtained by integrating segmentation and parsing. The serial model presented here, however, has the benefit of surveyability and modularity. The drawback of any morphological analysis program is over-acceptance; there are lots of compounds which are grammatical but which are semantically or pragmatically anomalous. These words are nevertheless accepted by the parser if they occur. In our case, the problem of overacceptance is diminished automatically by the mere fact that less morphological rules are used than in most systems. The remaining over-acceptance of compounds, however, cannot be completely solved at the present state-of-the-art of semantics and pragmatics. Another problem is the incompleteness of the dictionary, but we believe that this drawback can be removed by including a user dictionary (see Chapter 6).

3.3 Organisation of a Lexical Database

A lexical database (computer lexicon, dictionary) is a computer-readable list of language elements. For each entry (an element of the list), some relevant information is provided.

3.3.1 Design Choices and Problems

Traditionally (e.g. Geeraerts and Janssens, 1982), two aspects of dictionary organisation (for both printed and computer dictionaries) are distinguished: *macro-structure* (how do we collect a list of entries?) and *micro-structure* (which information is relevant for each entry?). A third design problem, relevant only to computer dictionaries, concerns the *storage* of the dictionary in memory, and the construction of *search* algorithms to find information quickly (how can we minimise access time?).

Macro-Structure. Some problems to be solved in the design of the macro-structure of a dictionary are the *kind* of language elements to be entered (morphemes, citation forms, word forms, phrasal idioms), the *number* of entries (the *n* most frequent, as many as possible[9]), and the *order* in which the items are put (alphabetically sorted or in order of decreasing frequency). The latter is different for printed and computer dictionaries.

[9] It is clear that a dictionary can never contain *all* words of a language, because the vocabulary is infinite in principle. E.g. the set of numerals is infinite because the set of numbers is infinite, and unpredictable neologisms and derivations (especially compounds) can be and are continuously formed.

Micro-Structure. In organising the information for each individual entry, the main problem is to determine which information should be available. Information which could be relevant includes: spelling form, spelling variants, phonological transcription, stress position(s), syllable boundaries, morphological boundaries, syntactic category, subcategories (e.g. transitive, intransitive, reflexive for verbs), selection restrictions (e.g. *to eat* takes an animate subject), case frames for verbs, semantic features, definitions in terms of synonyms, antonyms and hyponyms, origin (Romance or Germanic), pointers to a concept (for content words), written frequency, spoken frequency ... . Which information is necessary depends on the particular application or task at hand and whether we want to retrieve it or compute it. For example in *hyphenation*, we need information about the morphological boundaries in a word form (see Chapter 5), in *spelling correction* we need information about the morphological structure and the phonological representation (see Chapter 6). This information can be either stored or computed.

Search and Storage. We will give a short characterisation of the most important options which are available. Detailed accounts of different search, sort and storage techniques in general can be found in Knuth (1973) and, applied to natural language dictionaries, in Hays (1967).

A first design feature to be considered is the place of storage: internal or external memory. We presume that it is not feasible with present-day hardware to store a complete dictionary (typically involving tens of thousands of records) in internal memory. It is, however, possible to combine search in internal and external memory to achieve high performance.

The most straightforward way to organise a dictionary in external memory is to represent each entry as a record with as its key the spelling form and as fields the different types of information associated with it. These records can be ordered alphabetically, in order of decreasing frequency or a combination of the two. A *sequential search* consists of comparing a word looked for with all keys of the dictionary records until a match is found, or until all keys have been checked without a match, in which case the search fails. This method is unpractically slow.

Sequential search can be sped up considerably, however, by partitioning the dictionary and storing in internal memory a table each element of which points to the beginning of a partition in external memory. Such a table could consist of the set of first-two-letter strings of the keys in the dictionary (theoretically 26^2 elements for

Dutch and English, in practice much less). Each element of this table would contain a pointer to a partition in external memory containing the entries the keys of which start with these two letters (Figure 16).

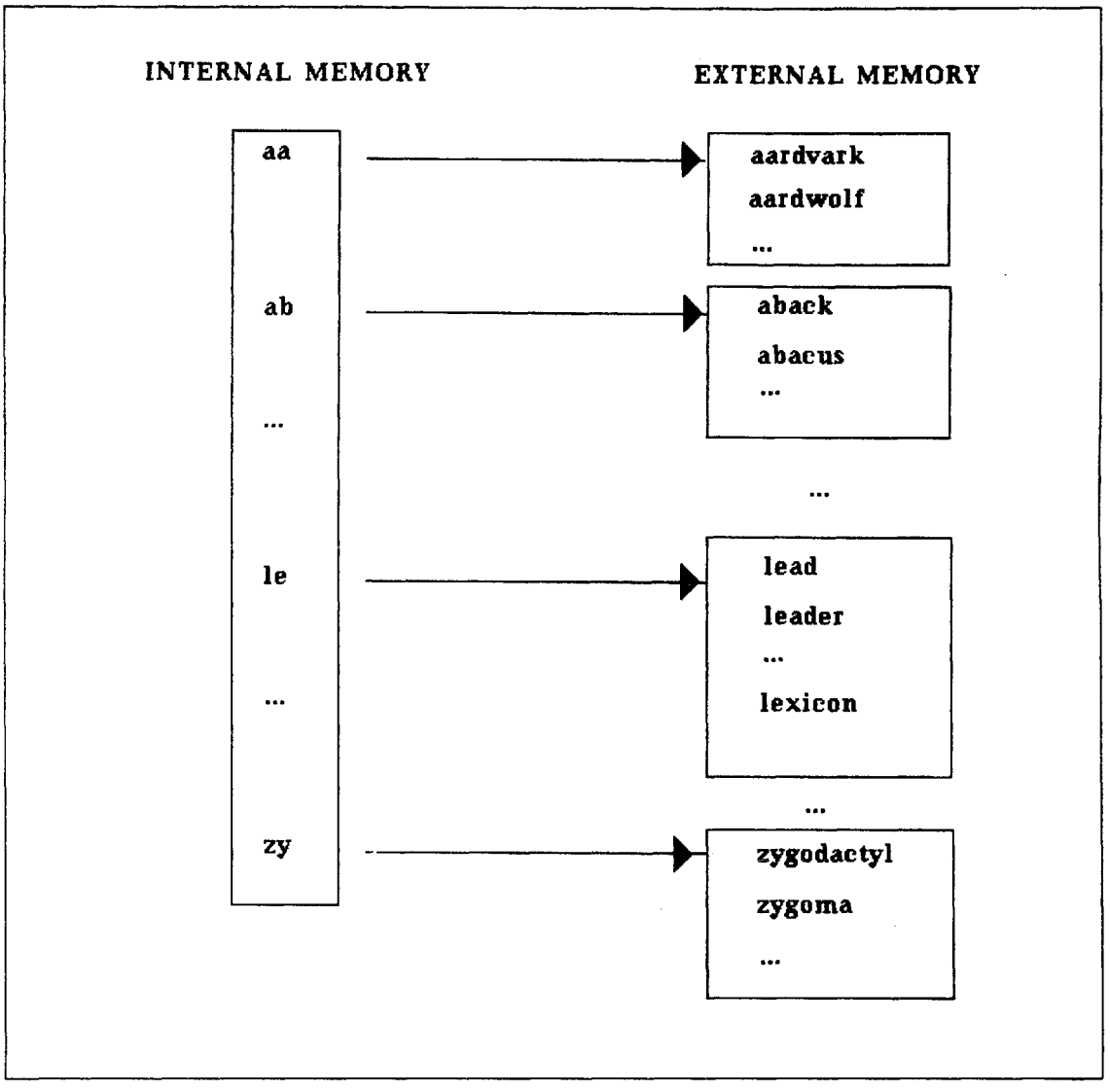


Figure 16. Indexed-sequential dictionary organisation (letter table in internal memory, partitioned dictionary in external memory).

E.g., if we look up the word *lexicon*, we look up *le* in the table in internal memory. There we find the external memory location where the first word beginning with *le* (*lead*) is listed. From that position on, we do a sequential search until either we find *lexicon* or we reach the end of the partition. This method has been termed *indexed-sequential* storage and search. We applied this method (or rather a variant of it) to the organisation of our main dictionary. The operating system VMS (Digital) provides standard programs to create indexed-sequential versions of sequential files.

Another combination of search in internal and external memory consists of storing the keys of all entries in internal memory to permit fast search. The keys contain a pointer to a unique memory location in external memory from which the relevant information can be retrieved. A technique which allows us to do this is storage in the form of a *letter table* combined with a *trie-search*. In trie-lookup, the searched word is matched to the keys one letter at a time. A letter table is an m-ary tree, with letters as nodes, and arcs pointing to possible following letters. Consequently, a key is interpreted as an ordered sequence of nodes, connected by continuation arcs and leading to a pointer to a location in external memory. In a complete letter table for a language, we would find at the first level all possible word-initial letters for that language (say 26). At the second level, 26^2 continuations are theoretically possible, at level three 26^3 etc., but restrictions on morpheme structure reduce the possible continuations to a fraction of what is theoretically possible. The letter table is therefore an efficient storage medium for large collections of words.

An example will make things clearer. Suppose we have a lexicon consisting of only the words *aap, aan, apen, appel, banaal* and *banaan*. A trie structure for this dictionary is given in Figure 17. In this figure, an asterisk (the 'leaf-mark') indicates that the sequence of nodes followed to reach it constitutes a word. A pointer to dictionary information can be added after the 'leaf-mark'. *Adding* words to the dictionary is done by following a path through the trie until there is no more match with the string to be added, at which point a new branch to the tree is inserted, with the remaining letters of the string. *Searching* in a trie is looking for a path through the tree. If one is found when all letters of the input string have been matched to nodes, and a leaf-mark is immediately following, then the input string has an entry in the dictionary, and the associated information can be retrieved by using the pointer after the leaf mark. If not all letters of the input string can be matched to nodes during a legal path through the tree or if no leaf-mark follows when all letters of the input string are matched, then the search fails. In the latter case (failure), a longest match is found, which may be helpful in some applications. It could be exploited, for example, in error correction (see Chapter 6).

A trie structure is also useful in applications where it is interesting to know the possible continuations of a string (the *cohort*). E.g. in our mini-dictionary the cohort of the string *ap* is the list *(apen appel)*. Cohorts can be straightforwardly computed in a trie structure. An example of such an application is a *typing aid* which completes words as they are being typed in. With each new letter typed in, the cohort of the

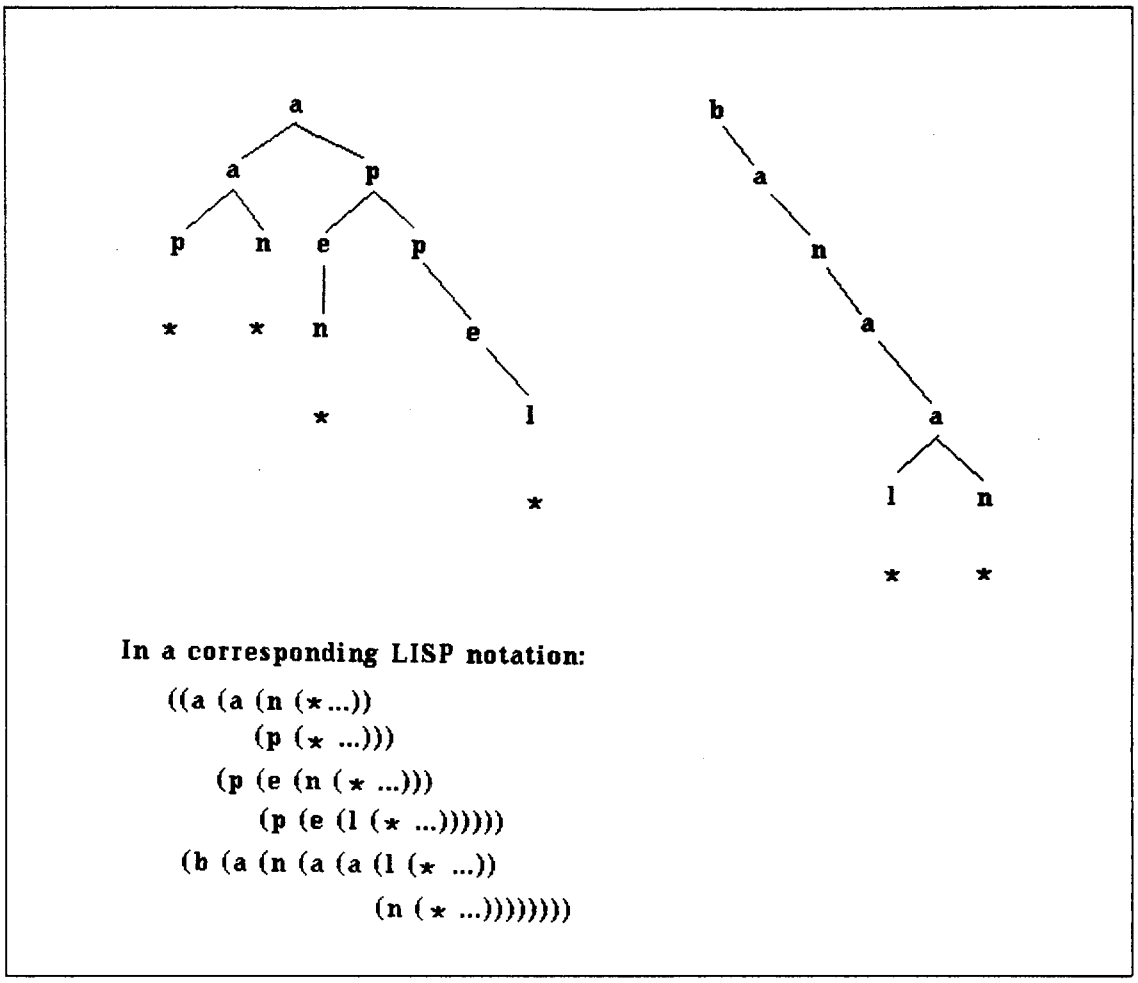


Figure 17. A trie structure for a mini-dictionary; tree-notation and Lisp list notation.

string obtained so far is computed. Whenever the cohort contains one element, this element is probably the word being typed, and the remaining letters can be printed by the program. This application seems particularly useful for helping motorically handicapped people. We used a trie-structure to implement a version of our user dictionary; a dictionary which can be filled and updated by the user of a particular application, and which can be combined with the main dictionary.

Still another method to search quickly in external or internal memory is *hashing*. Hash functions, which are mostly based on multiplication or division, compute a (preferably unique) memory location on the basis of (a numerical version of) the key. In searching, we simply apply the hash function to the searched word, and look at the location in memory indicated by the resulting value. An example of a hash function is the summation of the ASCII code (or any numerical code) of each letter in the

key modulo the largest prime number smaller than the number of entries of the dictionary (e.g., 997 for 1000 entries). We have implemented a version of our user dictionary using this storage technique.

Another way to achieve the necessary *compression* of keys to be able to store them in internal memory is by avoiding redundant use of characters. E.g. the words *occult, occultistic, occupancy, occupant* and *occupation* (together 44 characters) could be stored *occult, 6istic, 4pancy, 7t, 6tion* (only 25 characters, 43% saving). The numeral refers to the number of letters which should be taken from the preceding word. This compression method is a variant of the letter table discussed earlier.

3.3.2 A Flexible Dictionary System

In section 3.2.1, the traditional storage versus processing dilemma was laid out in relation to morphological analysis. Applied to morphological synthesis, three theoretically possible choices in the dilemma could be the following:

(i) Store only citation forms and compute all derived forms.

(ii) Store citation forms and irregular derived forms, and compute the rest (this solution would appear to be the most 'balanced').

(iii) Store all citation forms and derived forms (regular and irregular). Compute nothing.

Present day technology permits us to store enormous amounts of lexical data in external memory, and retrieve them quickly. Soon, 'dictionary-chips' and optical disk lexicons will be available, improving storage and search possibilities even more. In view of this evolution, the traditional debate about storage versus computation becomes irrelevant when applied to language technology. Natural Language Processing systems should exhibit enough redundancy to have it both ways. For instance, derived forms should be stored, but at the same time enough linguistic knowledge should be available to compute them if necessary. There is some evidence that this redundancy is psychologically relevant.

Two competitive psychological theories about the organisation of the mental lexicon exist (roughly comparable to the storage versus processing controversy). One states that it is possible in principle to produce and interpret word forms without resort to morphological rules except in the (special) case of newforms. We will call this the *concrete hypothesis* (e.g. Butterworth, 1983). This hypothesis implies that

morphological boundaries are represented somewhere. The alternative *abstract hypothesis* claims that in production and comprehension rules are routinely used. E.g. in production, an abstract representation *work+[past]* is transformed by rule into *worked*. Within this hypothesis, it can be claimed that concrete word forms (like *worked*) are or are not present in the mental lexicon. In the former case, the lexicon is largely redundant. This duplicated information (co-existing rules and stored forms) could be part of the explanation for the fuzzy results in most experiments aimed at resolving the concrete versus abstract controversy (Henderson, 1985). One example of this fuzziness can be found in the work of MacKay (1972, 1976). On the one hand, the fact that regular past tenses of verbs are produced faster than irregular past tenses seems to suggest that people apply rules of word formation in the routine production of (regular) past tense verbs. On the other hand, evidence from a rhyming task seems to suggest that concrete forms are nevertheless immediately available whether they are regular or not.

The problem of how much information should be present in a dictionary (both on the micro- and the macro-structure level) becomes irrelevant if we assume two types of dictionary: (a) a *mother dictionary* (MD) containing as much relevant information as can be gathered[10], and (b) various *daughter dictionaries* (DDs), tailored to suit a particular application and derived from the mother lexicon. We have called such a system a *Flexible Dictionary System*, and in section 8.2, we will give a detailed description of the architecture of a FDS, and how it can be constructed (semi-)automatically. Of course there are both practical and theoretical restrictions to the amount of information entered in the MD. Memory is not unlimited, so it would not make very much sense to store e.g. all inflectional forms of nouns and verbs when compiling a dictionary for Finnish. In general, it would not be interesting to store forms which can be analysed and generated by a simple exception-less rule.

3.3.3 The Top-10,000

The development of the Top-10,000 dictionary is a project carried out at the University of Nijmegen. A computer readable version of a dictionary containing some 10,000 citation forms (11,605 to be precise; class ambiguity in a citation form was resolved by entering a separate form for each class a particular citation form could belong to) with some syntactic information, compiled from various sources, was initially available.

Construction. A (function-oriented) program developed by Dik Bakker, Henk Schotel and the present author, generating a number of inflectional and derivational variants, was applied to this dictionary. The program was written in Franz Lisp, building upon an original Fortran program by Dik Bakker. It generates plural, diminutive and diminutive plural of nouns, all inflected forms of verbs, ordinal variant of numerals and inflected, comparative and superlative forms of adjectives. A computer readable Van Dale dictionary (Kruyskamp, 1982) was used by the program as a source of information about irregular word forms. After applying it, the dictionary was expanded by a factor five.

Although it works fairly well, we believe this particular program has a number of important drawbacks: first, no attention was paid to a user-friendly interface allowing the output of the program to be checked and corrected interactively; and second, the program drew upon an unrelated dictionary containing part of the information which had to be computed (the Van Dale). An object-oriented program such as the one described for the inflections of verbs (section 3.1) would be more suitable. The latter approach is more complete, more easily extensible and maintainable, more modular because of its object-oriented implementation, and it does not make use of external sources of information ('irregular' forms are to a large extent computed instead of simply listed). All relevant information about phonology, spelling and morphological structure is present in the program, and intervention by the user while checking and correcting the output is reduced to a minimum.

Apart from morphological synthesis, each of the morphological modules described earlier and the phonological ones in the next chapter, can be applied to extend the information present in the Top-10,000 dictionary: *morphological analysis* to find morpheme boundaries in wordforms, *phonemisation* to compute phonological representations of word forms and *syllabification* to indicate syllable boundaries. In section 8.2, the application of linguistic knowledge in lexicography will be discussed more systematically.

Organisation and Access. A sequential version of the Top-10,000 was transformed into an indexed-sequential file using standard operations available in VMS and UNIX-code developed by Eric Wybouw of the AI-LAB in Brussels. Access functions, retrieving specified records and fields of records in this indexed-sequential file were written as FORTRAN and C procedures, and can be executed from within a LISP environment. For the Lisp programmer, dictionary access operations take the

form of LISP functions (Figure 18 lists some of these).

```
(lookup 'word)          Looks up word and returns t (word is found) or nil
                        (word is not found). If a word is found, the record
                        containing it as a key is made the current lemma as
                        a side-effect.

(lemma:stress)          Position of word stress in current lemma.
(lemma:pronun)          Transliteration of current lemma into a phonological
                        alphabet.

(lemma:category)        One of noun, adj, adv, art, verb, prep, pro, conj,
                        interj, num.

(lemma:singularp)       True if current lemma is a singular noun.
(lemma:transp)          True if current lemma is a transitive verb.
(lemma:ordinalp)        True if current lemma is an ordinal numeral.
```

Figure 18. Some Lisp dictionary access functions.

Updating. It has already been mentioned that a dictionary can never be complete. It should be possible to add neologisms and newly derived forms to the MD in a convenient way. How this was done in our system will be fully explained in section 8.2. The modified or added records are stored in a user dictionary which acts as a shell around the MD. From time to time, the contents of this user dictionary is merged with a sequential version of the MD, and a new indexed-sequential version is computed (a procedure expensive in processing time).

3.3.4 Conclusion

The distinction between MD and DDs resolves the storage versus processing dilemma by introducing a possibly psychologically motivated redundancy to the dictionary system. The construction, extension and maintenance of the MD can be made more manageable by using the linguistic knowledge and processes, and optimal databases for specific applications (DDs) can be derived from the MD without much effort (see section 8.2). Current and forthcoming storage and search technology seem to guarantee that it is a feasible enterprise. With the Top-10,000 dictionary, we have a reliable database of the 10,000 most frequent citation forms and their derived forms, which could be a good point of departure for the development of a complete MD for Dutch using among other the linguistic knowledge described in sections 3.1 and 3.2 of this chapter, and in Chapter 4.

3.4 Related Research

In computational linguistics, much less work has been done on morpho(phono)logy than on syntax. This may be due to the fact that English, with its impoverished morphological system, has been the target language of most projects until recently. In these applications, all word forms (including inflected and derived ones) could be simply included in the dictionary with their phonetic representation. At most, related word forms (elements of the same paradigm) would be made to share common features, or some rudimentary form of affix stripping would be included to save storage space. It comes as no surprise then, that most original insights have come from research on languages with a rich morphology, e.g. Finnish. We will describe Koskenniemi's (1983) research on *finite state morphology* as well as two additional systems and compare them to our own approach. In addition, a number of related object-oriented approaches to morphology will be discussed. Finally, our model will be evaluated from the point of view of psycholinguistics.

3.4.1 Finite State Morphology

Recent computational morphophonology has been dominated by *finite state models*, as exemplified by the work of Kimmo Koskenniemi (1983, 1984) and Kay (1983). The model Koskenniemi proposes consists of a dictionary and a set of morphophonological rules.

The *dictionary* consists of a root lexicon and a number of affix lexicons. The latter are lists of inflectional and derivational suffixes. The root lexicon contains the *morphemes* of the language in their *underlying representation* (drawn from an alphabet of phonemes, morphophonemes, boundary symbols and diacritics), *syntactic and semantic properties*, and a list of pointers to continuation *lexicons*. If a particular lexicon is pointed at in a lexical entry, this means that all its members can co-occur with this morpheme. In a sense, these pointers determine the possible sequences of morphemes within a word.

Whereas rules in generative phonology are sequentially ordered, uni-directional and involving a series of intermediate stages; two-level rules operate simultaneously (in parallel), are bidirectional and relate the lexical level immediately to the phonetic or orthographic level, and vice versa (only two levels of description are necessary). The bidirectionality of the rules allows them to be used both in analysis and generation. The rules also act as a filter which blocks some morpheme strings allowed by

the continuation pointers in the lexicon. Consider as an example rule (1), taken from Barton (1985),

(1) $$\begin{matrix} y \\ i \end{matrix} <= \begin{matrix} = \\ = \end{matrix} \_\_ \begin{matrix} + \\ = \end{matrix} \begin{matrix} s \\ s \end{matrix}$$

which states that lexical *y* must correspond to surface *i* when it occurs before lexical *+s* (as is the case with *tries* which has lexical representation *try+s*). Two-level rules use character pairs as units instead of characters. The equality sign is a wildcard character. The arrow indicates that the character pair on the left must occur in the context of the character pairs on the right.

Each rule can be compiled (by hand or automatically) into a Finite State Transducer. An FST is basically a Finite State Automaton, but with two tapes instead of one. The transducer starts from state 1, and at each step, it changes its state as a function of its current state and the character pair it is scanning. Each FST should be interpreted as a constraint on the correspondence between lexical and surface (phonological or graphemic) strings. The FST corresponding to rule 1 is given in Figure 19. The use of . instead of : after state numbers indicates possible final states.

	y	y	+	s	=	(lexical)
	i	y	=	s	=	(surface)
state 1:	2	4	1	1	1	
state 2.	0	0	3	0	0	
state 3.	0	0	0	1	0	
state 4:	2	4	5	1	1	
state 5:	2	4	1	0	1	

Figure 19. Finite state transducer table for rule (1), adapted from Barton, 1985.

Different FSTs can either be collapsed into a single FST, or applied simultaneously. In analysis, the tape representing the lexical representation is written (with the additional help of dictionary stored in the form of letter trees, see section 3.3), and the tape with the spelling or phonological representation is read. In synthesis, the phonetic or spelling tape is written, and the lexical representation tape is read. It is obvious that such a system is both extremely efficient[11] and computationally attractive. The approach has been applied successfully to Finnish, and is being adapted for

[11] Nevertheless, it has been proved by Barton (1985) that analysis and generation with FSTs is an NP-hard (i.e. computationally difficult) problem in the general case. The computational class NP consists of problems which can be solved in polynomial time on a nondeterministic Turing Machine.

Japanese, Rumanian, French, Swedish, English and other languages (see Gazdar, 1985, for full references).

The approach does not endanger the usefulness of our own model, however. To begin with, Koskenniemi's system is devoted more to phonology than to morphology (only inflectional morphology is discussed, and alternations of longer sequences of phonemes are resolved in the lexicon). It would be difficult in his approach to account for restrictions on the combination of roots into compounds, for example, and the inclusion of rules modifying word internal structure (Ablaut, reduplication, infixation) would be problematic, too, as was noted in Golding and Thompson (1985).

Second, we have been working on quite a different level of abstraction. From the onset, we have wanted our system to be formalism- and theory-neutral. This implies, for example, that it should be equally well equipped to handle feature bundles (like in generative phonology) as monadic phonemes (Koskenniemi's approach). The rationale behind this was that enough problems remain in the organisation of a word formation component for Dutch to justify a flexible, open-ended system.

Furthermore, recent trends in phonological theory (*auto-segmental phonology* and *metrical phonology*) cannot be straightforwardly incorporated into a two-level approach, as the latter has only limited context-sensitivity, while this incorporation constitutes no problem for our own model.

Perhaps the most important difference is the fact that Koskenniemi starts from a lexicon in which underlying representations of morphemes are already present (his main interest is to construct an efficient parser and generator). In contrast, we tried to organise our knowledge base in such a way that the underlying representation can be *constructed* if needed.

3.4.2 Oracle

Oracle (Honig, 1984) is a general morphological analysis program (in principle language independent). It resembles our system in the independence of lexicon, morphological rules and spelling rules. The segmentation procedure is left to right, longest left first (as opposed to our longest right first), and parsing is interleaved with segmentation (as opposed to our sequential approach). The regular lexicon contains only categorial information. Entries of the irregular lexicon are listed with their

internal structure. An irregular lexicon was necessary to prevent a large number of invalid results by applying unproductive morphological processes routinely. This problem led us to a solution in which word forms are stored instead of morphemes. Morphological rules are described using an extended context-free grammar; spelling rules and context rules use a regular expression formalism.

As the program expects well-formed input, a large number of invalid word forms are accepted (and analysed). For example, both *koppen* and *kops* are accepted as the plural of *kop* (cup), while only the former is correct. Spelling rules undo those root modifications which produce spelling forms different from the lexical form. Like in our own system, Oracle incorporates no semantic processing. A superficial syntactic analysis of the sentence in which a word occurs is included to disambiguate between multiple analyses.

As we did not focus our research on analysis (in fact, we use some kind of analysis-by-synthesis approach), it is difficult to compare both systems. We believe that our analysis of spelling rules is linguistically more relevant, and our inclusion of a phonological component (next chapter) improves completeness and descriptive adequacy. To be fair, it should be mentioned that Oracle was not intended as a linguistic model, but it remains to be seen whether the formalisms developed are flexible enough to make the modeling of experimental linguistic theories a straightforward matter.

3.4.3 Lexicrunch

The Lexicrunch program (Golding, 1984; Golding and Thompson, 1985) induces rules of word formation given a corpus of root forms and the corresponding inflected forms, thereby compressing the dictionary. The resulting rules can then be used in analysis and generation. This approach is diametrically opposed to our own, which uses inflectional rules to expand a citation form into a word form dictionary, and which restricts morphological analysis to compound analysis including, whenever necessary, detection of internal word boundaries.

The basic operation which rules can perform is string transformation (replace substrings and concatenate strings). Rule induction proceeds in two stages: *data entry* and *rule compression*. During data entry, pairs of roots and corresponding inflected forms are accepted, and the system tries to find a set of minimal string manipulations which transform the former into the latter. The resulting, inefficient, rules are

subsequently compressed, which is an expensive operation. This process involves restating of transformations (there are lots of ways to achieve a particular string transformation), cross-classification of words, elimination of redundant transformations, and building a decision tree. Currently, the system has been applied 'adequately' to the past tense inflection in English, the first person singular indicative of Finnish and the past participle in French.

A major drawback to the system is that the inferred rules are linguistically nonsensical. They are based on the surface properties (appearance) of the input strings, and linguistically relevant representations like morphological boundaries, syllable structure and phonological representations are either absent, or remain unused. Therefore, the system seems applicable only as a means of dictionary compression, which, as we have argued earlier, is unnecessary in view of current storage and search technology, especially for languages like English and Dutch. Moreover, other dictionary compression methods exist which are computationally less demanding. The system can certainly not be used to model morphological competence and performance. On the other hand, we think the system is extremely useful in the linguistic analysis of *irregular* past tenses. The elements of the different Ablaut categories indeed seem to be related by appearance (both consonantal and vocalic properties of the stem): e.g. *drink, shrink, sink, stink*. The same goes for Dutch. E.g. *stinken, drinken, zinken*. Although this regularity is nicely captured by Lexicrunch, it does not justify its application to regular processes.

As regards Dutch, (concatenated) compound analysis is impossible with Lexicrunch, and several morphological rules which rely heavily on phonological information cannot be induced reliably on the basis of the spelling string alone (cp. section 3.1).

3.4.4 Other Object-Oriented Approaches

In this section, some recent approaches to linguistic description with an object-oriented tenor are described and compared to our own approach.

In *word grammar* (Hudson, 1984; Hudson and Van Langendonck, 1985) a language is represented as a network of linguistic *entities* (comparable to concepts), related by a limited number of *propositions* (comparable to subjects and type hierarchies). Linguistic entities are words, parts of words, strings of words, etc. Basic propositions are *composition* (relating a word to its parts), *model* (relating a more

general entity to its specialisation, this proposition makes selective inheritance of information possible), *companion* (relating a word to the words which it occurs with), *referent* (connecting words to semantic structure), and *utterance-event* (linking a word to the pragmatic context via uttering it). The similarity to our approach lies in the uniformity of representation (all linguistic knowledge is represented uniformly) and the object-oriented flavour. The main difference is that in Hudson's system all representation is declarative (descriptive) so that the use of the knowledge in generation and analysis is not considered.

De Smedt (1984) describes advantages of the object-oriented programming paradigm for the representation of syntactic knowledge. An object-oriented implementation of IPG (a psycholinguistic theory of sentence generation; Kempen and Hoenkamp, forthcoming) using the object-oriented language CommonORBIT is outlined in Kempen and De Smedt (in preparation).

In Steels and De Smedt (1983) linguistic structures are viewed as collections of descriptions (frames). Frames are organised into tangled inheritance hierarchies. Linguistic processing (building structures in analysis or generation) happens by applying frames and searching the hierarchy in both directions (i.e., generalisation and specialisation). The theory can be easily implemented using an object-oriented language.

Functional Unificational Grammar (KAY 1979, 1985) is another recent linguistic theory which lends itself to an object-oriented implementation. A functional description, which is the basic representation unit of FUG, is a set of feature-value pairs in which the values can be functional descriptions in their own right (this is basically a frame-structure). An implementation of functional descriptions as KRS concepts with subjects as features and concepts as fillers is straightforward. Unification can be defined as a subject of the concept representing the *functional description* type.

3.4.5 Psycholinguistic Research

In Chapter 1, it was claimed that a computer model should be constrained by both linguistic and psychological theory. In this section we will review our model from the point of view of related research in psycholinguistics. What we see happening in the domain of morphology, is a convergence of psycholinguistic and computational efforts. In both disciplines the construction of explicit models of morphological and lexical processing and representation is attempted. Even the terminology overlaps

(e.g., *lexicon file, computational order, on-line processing, affix-stripping*).

A number of observations and experiments suggests that internalised morpho(phono)logical rules are a psychological reality[12]. This evidence can be divided into two categories: *production of errors* (overgeneralisation in child language: Schaerlaekens, 1979; speech errors by adults: Fromkin, 1973 and Garrett, 1975 and aphatic speech: Henderson, 1985), and *production or comprehension of new complex words* (rule-governed creativity in adults and children: Berko, 1958). All error forms and new forms show *phonological accomodation*, i.e. the correct allomorph and pronunciation are produced in the phonological context. These data seem to suggest that a relatively independent morphological level exists between the syntactic and the phonological levels (Gibson and Guinet, 1971; Laudanna and Burani, 1985; Cutler, Hawkins and Gilligan, 1985). Units at this level are *morphemes* (stems) and *affixes*, *word forms* and *rules*. It is exactly these units which are also represented as concepts in our computational model.

Most of the data discussed so far, however, give only a vague idea about the morphological processing going on in language users and about the memory representations they use to store lexical material. What we would like to know is whether morphological rules are routinely used in production and comprehension, or only in special cases. Furthermore, we would like to have some information about how lexical entries are represented and how they are accessed, and whether inflections and derivations are processed differently.

Unfortunately, psycholinguistic research does not help us to resolve these problems. Most of the more detailed theorising on the basis of experiments is highly controversial (see Henderson, 1985 for a recent discussion). This controversy may be due in part to the debatable precision of a psycholinguist's instruments (lexical decision task, priming, naming tasks). To begin with, there is a serious risk of *task effects* (strategies generated by the task or the context, but without correlate in 'normal' processing). Furthermore, only strictly serial models and theories can be tested. Different time lapses between input and output of a black box have nothing to say about the organisation of the black box if *parallel processing* is allowed. Worse, it is

[12] Notice that this assurance does not extend to the way these rules are psychologically (or biologically) implemented. There may be ways that rule-governed behaviour can be produced in which there is no explicit representation of any rules (by parallel distributed processing models, for example, see Rumelhart and McClelland, 1986).

nearly impossible to balance all possibly relevant independent variables in the selection of the test items (it is simply not clear what would constitute an exhaustive list of relevant dependent variables in morphological processing). Even the selection of the test material implies a theoretical choice (e.g. which words are affixed and which are not?). In view of these provisos, it is only natural that not much agreement has been achieved among workers in this field. Another possible reason for the lack of clear results — the fact that derived and inflected forms may be at the same time stored and computed — was discussed in section 3.3.

Although our morphological model was primarily designed from a technological point of view, it could be adapted to serve as a psychological model of human morphological processing. The level of abstraction (concepts for affixes, morphemes, word forms, paradigms and rules) seems empirically right, and theories like the *morphological decomposition model* (Taft and Forster, 1975, 1976; Taft, 1981), *the logogen model* (Murrell and Morton, 1974; Morton, 1970, 1979a, 1979b) and the *cohort model* (Marslen-Wilson and Welsh, 1978; Marslen-Wilson and Tyler, 1980; Marslen-Wilson, 1980) can be easily simulated using the same concepts (but with additional subjects).

In a sense, computational models are richer than psycholinguistic theories because they have to be explicit to the extreme. In general, a lot of problems which we are confronted with in the development of artificial morphological generators and analysers (ambiguity, backtracking, heads of compounds, Ablaut, root structure changes etc.) are almost never mentioned in psycholinguistic research. Perhaps language technology can provide a fresh perspective on psychological theory building in these matters (rather than vice versa).

3.4.6 Conclusion

Summing up this brief overview of related research, we believe that our model has some advantages setting it apart from other approaches. The basic open-endedness and extensibility inherent to object-oriented implementations make the system ideally suited as a 'laboratory' for the studying of different linguistic formalisms, models and rule interactions (this will become even clearer in the next chapter). Psycholinguistic models can also be implemented and evaluated with it. At the same time, the implemented program has proved to be efficient enough to be used in concrete practical applications (see Part III).

However, our model may be inferior in computational efficiency and elegance to finite state approaches. It would therefore be interesting to investigate whether Dutch morphophonology can be described similarly, once we have agreed upon the relevant rules and their interactions.

CHAPTER 4

Aspects Of Dutch Phonology

In this chapter, an object-oriented database of phonological knowledge for Dutch will be developed, and two phonological processes which make use of this knowledge (phonemisation and syllabification) will be discussed in detail. In section 4.1, the syllabification algorithm is described, and the role of internal word boundaries in this process is explicated. In section 4.2, the phonemisation algorithm is outlined, and the architecture of our phonological database and its user interface are sketched. Throughout the chapter, special emphasis will be put on the interaction between phonological and morphological knowledge and processes.

4.1 A Syllabification Algorithm[13]

4.1.1 The Syllable as a Phonological Unit

In general, the Dutch syllable conforms to the *sonority hierarchy* (Kiparsky, 1979), a universal restriction on the build-up of syllables. In essence, this condition specifies that the sonority of phonemes decreases from the inside of the syllable to the outsides, following a sonority hierarchy in which vowels are most sonorant, followed by semi-vowels, liquids, nasals, fricatives, and plosives: — in that order. In Dutch, syllable-initial clusters like *sp*, *st* and *sk* ignore the hierarchy.

In addition to the sonority hierarchy, language-specific restrictions on length and form of syllable-initial and syllable-final clusters, and on the nucleus of the syllable can be defined. E.g. in Dutch, only vowels can be *syllabic* (be the nucleus of a

[13] This section is based in part on Daelemans (1985c).

syllable), syllable-initial clusters cannot be longer than three phonemes, etc. Generally, there are two complementary ways to obtain these language-specific data: one more bottom-up, the other more top-down. A phonologist would try to describe morpheme and syllable structure conditions by means of positive and negative rules, filters and other constructs (e.g. Trommelen, 1983 and Booij, 1981 in the framework of generative phonology). This approach is based on empirical data, but has a top-down (deductive) nature. The complementary approach is more bottom-up (inductive) in that it would involve statistical analysis on a large quantity of data. In the latter case, we are confronted with a vicious circle: we need a syllabification program to produce the raw material for our statistical analysis (i.e. syllables) and we need the statistical information to make a proper syllabification program.[14]

While the deductive approach is more pleasing from a linguistic point of view, statistical analysis is needed because some clusters which are pronounceable on theoretical grounds may not be realised in the vocabulary of the language. On the other hand, there is a large amount of loan words in Dutch. Some of these may contain un-Dutch clusters. Nevertheless, we wish to divide them correctly into syllables since they are not felt to be foreign any more. The lists we will use are based on a statistical study of the spelling syllable by Brandt Corstius (1970) (for which he used his SYLSPLIT hyphenation algorithm) and on Bakker (1971). In the process of testing our program, the lists were considerably modified and extended, according to the empirical data.

A useful description of the syllable, borrowed from *metrical phonology* is the one in Figure 1. This diagram suggests that a syllable is a string of phonemes, consisting of a *nucleus* (a single vowel: short, long or diphthong), optionally preceded by consonants (the *onset*) and/or optionally followed by consonants (the *coda*). It can be shown that the *rhyme* is a phonologically relevant unit by the fact that restrictions exist on the co-occurrence of nucleus and coda without comparable restrictions on the co-occurrence of onset and nucleus (Figure 2, adapted from Booij, 1981 and Trommelen, 1983).

Another traditional argument for the phonological relevance of the rhyme are rules like *schwa-insertion*, which would have the rhyme as their domain (Trommelen,

[14] The problem disappears if we entertain the idea to collect a large enough corpus of syllables *by hand.*

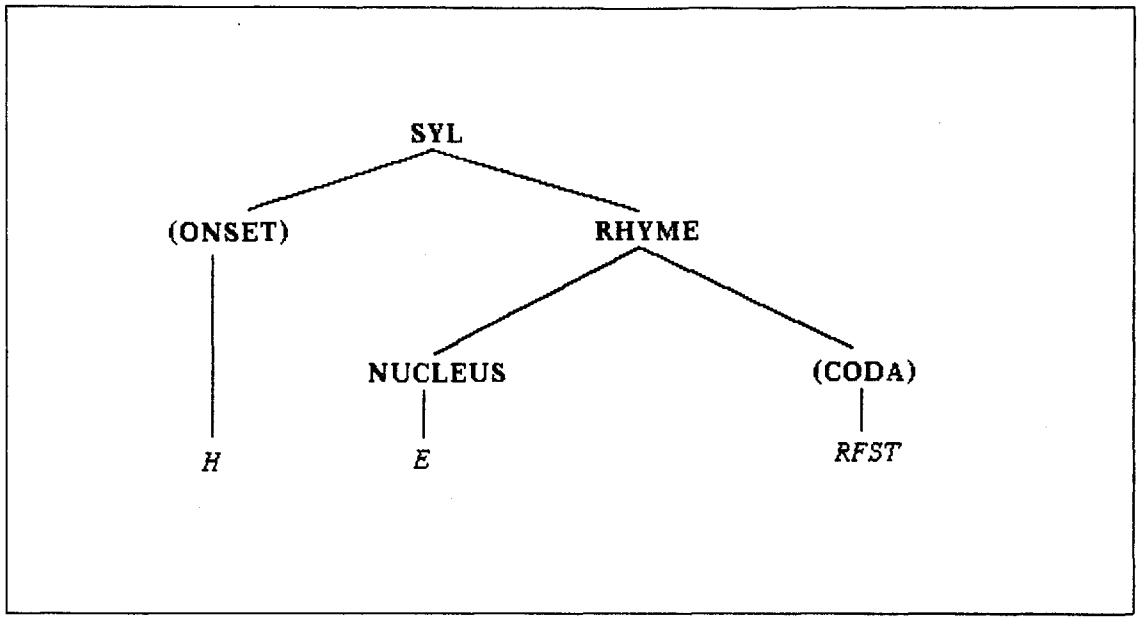


Figure 1. Syllable Structure. Parentheses indicate optionality.

1. Long vowels or schwa cannot be combined with the velar nasal to form a rhyme.

   *aang, *eeng, ...

2. Long vowels or schwa cannot be combined with [b] to form a rhyme.

   *aab, *eeb, ... (Exception: *foob*)

3. Diphthongs and [r] cannot be combined to form a rhyme.

   *eir, *aur, ... (Exception: *taur*)

4. If the nucleus of a rhyme is stressed and long , the coda must be of the form (C)(s)(t).

   *oemp, *aalm, ...

Figure 2. Restrictions on the Rhyme of Dutch Syllables.

1983). Schwa-insertion is a rule which inserts a schwa between a liquid and a non-coronal or nasal consonant if they occur in the same coda.

The boundary between two syllables may be fluid in some cases, but in general, speakers of a language can indicate syllable boundaries without problem, which makes syllabification a competence phenomenon. The phonological process of splitting up a word into syllables is governed by a universal *maximal syllable onset principle* (Selkirk, 1982). This principle dictates that in a consonant cluster between two

nuclei, as many consonants as possible belong to the same syllable as the following nucleus. The remaining consonants belong to the previous syllable. Again, Dutch does not follow this principle completely. A contradictory rule which states that short vowels occur in closed syllables forces the syllabification of words like *pastei* (paste) to be *pas-tei* instead of the expected *pa-stei*.

The algorithm we will describe in the next section assigns syllable boundaries to spelled words. As with the morphological module in the previous chapter, spelling was our point of departure for practical reasons: most language data are accessible to the computer via spelling exclusively. Apart from this, the algorithm was designed with practical applications (like hyphenation) in mind, which makes spelling an obvious choice.

4.1.2 A Computational Approach to Syllabification

Automatic syllabification remains a thorny problem for computational linguistics. The rules, which are simple for humans, are difficult to implement because they make extensive use of knowledge about the internal structure of words (morphological knowledge). In this chapter, new statistical data necessary for developing syllabification algorithms are presented, and a system is outlined which uses a lexicon and a word form parser to imitate the capacity of language users to analyse words morphologically. The presence of a lexicon suggests a simple solution to the problem: store the syllabification of each word form explicitly in the lexicon, and retrieve it when needed. However, we need a program to do this automatically for existing lexicons, and to cope with new words.

The algorithm, based on the approach taken by Brandt Corstius (1970), but with important modifications, splits up Dutch words into spelling syllables, following an explication of the rules provided by the official Woordenlijst van de Nederlandse Taal (word list of the Dutch language), and was inspired by recent (generative) phonological research on the subject. Phonological syllables do not always completely coincide with spelling syllables, but the latter can be easily derived from the former. Apart from its application as an automatic hyphenation program of Dutch text (section 5.2), the system plays an important role in the grapheme-to-phoneme conversion system described in section 4.2. This system needs information about the syllable structure of words. An overview of a computational theory of syllabification is given in Figure 3. In the remainder of this section, we will discuss the different parts of this

scheme in detail.

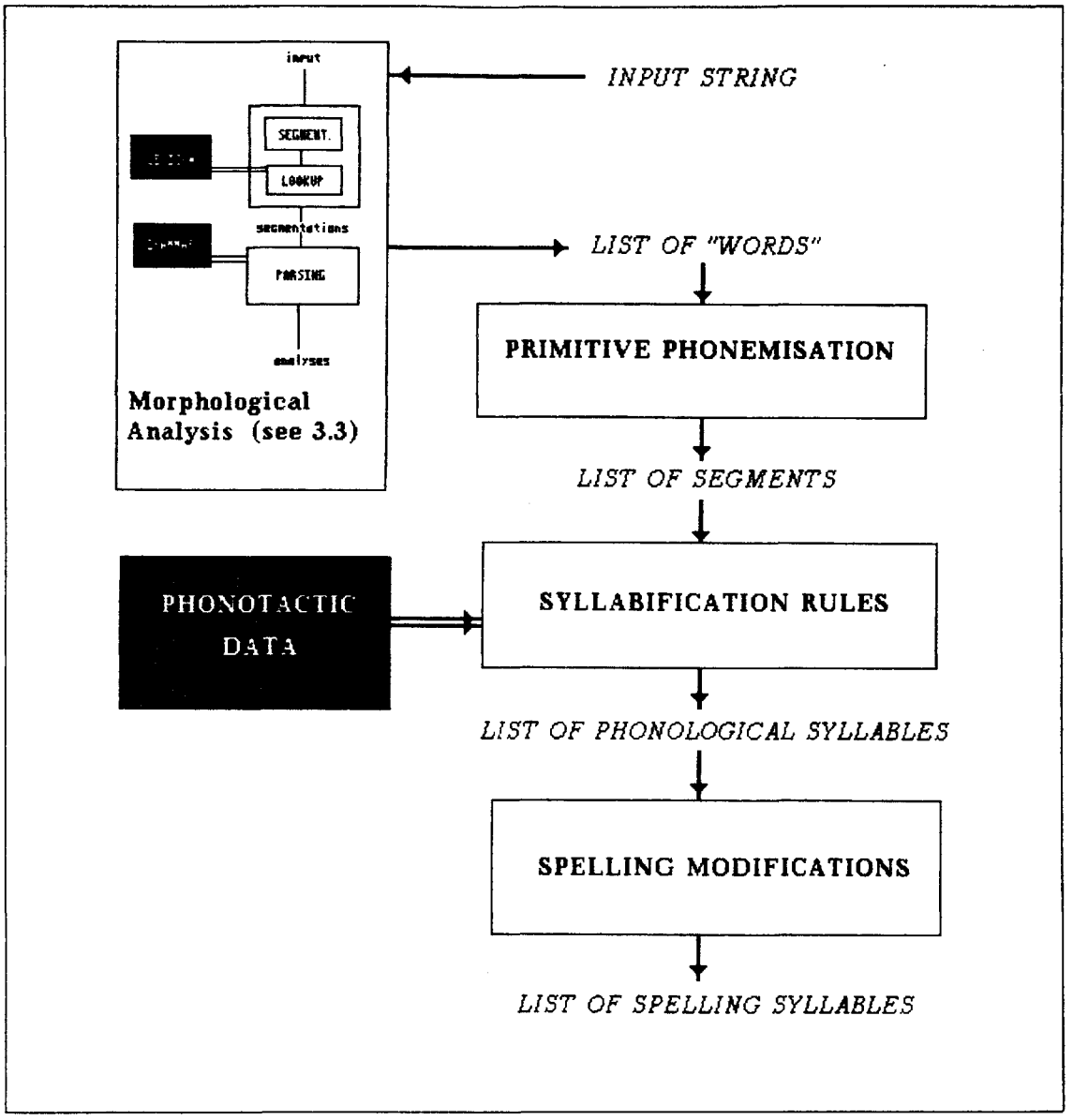


Figure 3. An outline of the syllabification algorithm. Black boxes represent data, white boxes processes. The double arrows indicate which data are used by which processes,. The single arrows indicate the input and output relations between representations and processes.

The processes are KRS subjects, attached to the word form concept described in the previous chapter. This implies that, although syllabification rules apply after morphological synthesis at the lexical level, syllable representations are accessible to the spelling filter attached to the spelling subject of the word form concept.

During *morphological analysis*, morphological word boundaries are inserted into a string of spelling symbols by a parsing process (section 3.2) which uses a word list (lexical database, see section 3.3). In the *primitive phonemisation* phase, each 'word' (a string of spelling segments between word boundaries) is transformed into a string of vocalic and consonantal segments. We have called this process primitive phonemisation because, in this stage, a first step from spelling to sound is taken. *Syllabification Rules* apply to the output of the segmentation stage. They are based on the Maximal Onset Principle, but with important extensions and modifications for Dutch. The result of this stage is a string of *phonological syllables*. An additional set of spelling modifications is necessary to transform these into spelling syllables.

4.1.2.1 Monomorphematic Words

In this section we concentrate on the division into syllables of words without morphological structure and words in which morphological structure has no bearing on syllabification (i.e. after morphological analysis). The rules described in this section therefore not always work in the case of complex words.

The (more or less) language-independent *process* we will describe is to be supplemented with language-specific *data*, as described earlier. These data include lists of the syllabic and non-syllabic segments of a language and lists of possible onsets and codas of Dutch syllables. For Dutch, a *syllabic segment* is defined as a spelling symbol or a string of spelling symbols pronounced as one vowel or diphthong (a, ee, ij, ooi, ui, ...). Each syllabic segment is the nucleus of a syllable. Since each syllable has exactly one such nucleus, and syllabic segments can only function as the nucleus of a syllable, the number of syllables of a word (a useful piece of information for some applications, notably style checking) can be measured by counting the number of syllabic segments in it (cp. Brandt Corstius, 1970). A *non-syllabic segment* is a spelling symbol or string of spelling symbols pronounced as one consonant or otherwise analysable as a consonantal unity (p, t, w, ch, qu, ...). Table 1 lists all segments for Dutch.

The clusters *ph*, *th* and *ng* are treated as a single segment. However, they are removed from the table in some applications (e.g. an hyphenation algorithm without morphological analysis, 5.2.3.1). In the latter applications, they must be treated as two-segment clusters, due to possible confusion in words like *uit-houden* (to suffer), *op-houden* (to stop), and *in-gang* (entry).

Syllabic segments:

a, e, i, o, u, y, aa, ee, ie, oo, uu, ae, au, ij, ei, eu, ai, oi,
ou, oe, ui, oy, ay, ey, uy, aai, aau, eeu, eui, ieu, oei, ooi,
eau, oui, oeu, oey, aay, ooy.

Non-syllabic segments:

b, c (pronounced /k/ or /s/), d, f, g, h, j, k, l, m, n, p,
qu (pronounced /k/ or /kw/), (ph, th, ng)
r, s, t, v, w, x (pronounced /ks/), y (as in *yoga*), z, ch.

Table 1. Syllabic and non-syllabic segments of Dutch.

The first step in the algorithm consists of transforming the input word (a string of spelling vowels and spelling consonants) into a string of syllabic and non-syllabic segments (*primitive phonemisation*). E.g.:

schreeuwen (to cry) is segmented *s-ch-r-eeu-w-e-n*

apparatuur (apparatus) is segmented *a-p-p-a-r-a-t-uu-r*

To obtain this result, a function is used that separates spelling vowels from spelling consonants, and a function which analyses adjacent strings of spelling vowels or consonants from left to right until the longest possible segment is found. E.g. the vowel string *aaie* in *papegaaien* is analysed as *aai-e*, and not as *aa-ie*, because *aai* is the longest possible syllabic segment which can be found, passing through the string from left to right.

This particular strategy to parse vowel clusters is possible because in general in Dutch, a diaeresis or hyphen is used when the division 'longest possible left part + rest' is not correct. E.g. *zoëven* (a moment ago), *auto-ongeluk* (car accident). The diaeresis and hyphen prevent the analysis of *oe* and *oo* as one syllabic segment. This permits us to use a *deterministic vowel string parser* even in morphologically complex words (see next section and also section 5.2.3.1).

There are some exceptions to this strategy; the vowel strings *iee*, *aie* and *oie* are always divided *i-ee*, *a-ie* and *o-ie* (longest right part). E.g. *materi-eel* (material), *moza-iek* (mosaic), *zo-iets* (something like that). When the other division is the proper one, this is signalled by a diaeresis. (e.g. *drieëndertig*, thirty three). Furthermore, *ieu* is only interpreted as a single segment if a *w* follows immediately (e.g. *nieu-we*, new). If this is not the case, the analysis is *i-eu*, as in *superi-eur* (superior).

Alternative solutions to this problem exist. In an approach used by Boot (1984) a string of spelling vowels is assigned syllable boundaries by looking for patterns in it in a well-defined order. This order was determined by the general principle *'longest patterns first'* and by empirical results (exhaustive testing). Basically the same idea was put forward by Wester (1985a, 1985b) in the context of a generative description of the function of diaereses in Dutch spelling. We will comment upon the general approach by means of the proposal of the latter. Wester argues correctly that the traditional view of the diaeresis as *'written on the second of two adjacent (spelling) vowels which together can be read as one vowel, but should not be read as such'* is deficient in that this rule would require a dramatic increase in the use of diaereses. For example, diaereses would have to be written in the following words: **geüit* (uttered), **ingeniëur* (engineer), and **bloeïen* (to blossom). Wester explains this fact by postulating an ordered list of rules (Figure 4), rewriting strings of spelling vowels to segments (cp. the ordered list of patterns in Boot, 1984)[15].

```
(1)     [+Voc]i[+Voc]i -> <+Voc,+Voc>
(2)     [o][e][i]      -> <oei>
(3)     [e][i]         -> <ei>
(4)     [u][i]         -> <ui>
(5)     [e][u]         -> <eu>
(6)     [a][u]         -> <au>
(7)     [o][u]         -> <ou>
(8)     [o][e]         -> <oe>
(9)     [i][e]         -> <ie>
(10)    [+Voc]         -> <+Voc>
```

Figure 4. Ordered list of rules rewriting strings of spelling vowels to segments. Square brackets indicate spelling vowels, angular brackets syllabic segments. Based on Wester (1985a).

A diaeresis is only necessary when the lexical representation of a word does not coincide with its mechanical reading (as determined by applying the rules in Figure 4. Examples are: *geëist* (demanded; lexical = *ge#eist*, mechanical = *gee-ist*), and *kippeëi* (chicken egg; lexical = *kippe#ei*, mechanical = *kippee-i*). In the earlier cases a diaeresis is not necessary since both readings coincide: *geuit* (uttered, lexical = mechanical = *ge-uit*). If the Boot-Wester hypothesis is correct, vowel strings can always be provided a correct syllable-boundary by means of the list of rewrite rules and the information supplied by diaereses. Apart from being obviously incomplete

[15] Incidentally, the two ordered lists are completely different and contradictory. Many examples given by Wester cannot be solved properly by means of Boot's list.

(*ij*, *ooi*, *aai*, *eeu* and others should be added to the list), a problem exists for the vowel string *ieu* which can be divided (*ingeni-eur*, engineer) or not (*nieu-we*, new) depending on the context. While a left-to-right parsing algorithm can solve this problem by adding a context-sensitive rule, as we showed earlier, there is no way to prevent the incorrect division *nie-uwe* in Wester's system (or when *ieu* is added to the list, the incomplete division *in-genieur*). While equally adequate in principle, we see therefore no reason to prefer the Boot-Wester approach to our own.

After this preliminary analysis into segments, traversing the string of segments from left to right, each cluster of one to seven non-syllabic segments intervening between two consecutive syllabic segments is provided a syllable boundary. In our examples:

s-ch-r-eeu-w-e-n

eeu, e --> compute syllable boundary for cluster w

apparatuur

a, a --> compute syllable boundary for cluster pp

a, a --> compute syllable boundary for cluster r

a, uu --> compute syllable boundary for cluster t

It follows that word-initial and word-final non-syllabic clusters are never split (they do not occur between two syllabic elements of the same word).

The following set of rules is used to distribute the non-syllabic cluster over two successive syllabic segments.

Rule 1. *If there are no non-syllabic segments (the cluster is empty), insert a syllable boundary between the syllabic segments.*

E.g. *moza-iek* (mosaic), *cha-os*

If there is a diaeresis in spelling, it can always be substituted by a syllable boundary before the element which carries it. We assume that a diaeresis is present whenever it is prescribed by the spelling rules of Dutch. E.g.

Israël --> *Isra-el*

naäpen (to ape) --> *na-apen*

Rule 2. *If the cluster consists of one non-syllabic segment, insert a syllable boundary before it.* E.g.:

la-chen (to laugh), *po-ten* (paws), *stra-ten* (streets)

Rule 3. *If the cluster consists of two non-syllabic segments, and it belongs to the set of cohesive clusters, insert the syllable boundary before it. Otherwise insert it between the two non-syllabic segments.* E.g.:

pot-ten (pots), *kan-ten* (sides) versus
li-vrei (livery), *a-pril*

Cohesive clusters (Table 2) contain two inseparable non-syllabic segments.

vr, vl, (th, ph), sch, pr, br, tr, dr, cr, kr, gr, fr, pl, bl, cl, kl, fl, kw E.g.: *li-vrei, me-thode, logi-sche, a-pha-sie* (*ph* is an obsolete spelling of /f/), *a-pril, ze-bra, ma-troos, Ma-drid, a-cryl, ma-kro, a-gressie,* *A-frika, di-ploma, pu-bliek, cy-clus, re-klame, re-flex, reli-kwie.*

Table 2. Cohesive clusters in Dutch.

Sometimes they are inseparable because the first segment of the cluster cannot occur in syllable-final position (*vr, vl*), sometimes because the cluster is pronounced as one segment (*th, ph*, sometimes *sch*). In the remaining cases, the clusters occur in loan words. They then consist mostly of a plosive, followed by a liquid. Note that *th* and *ph* are only present in this table if they are not present in Table 1.

Rule 4. *If the cluster consists of three or more non-syllabic segments, then traverse the cluster from right to left, and find the largest possible cluster which can function as the onset of a syllable in Dutch (i.e. which belongs to the set of possible syllable-initial clusters; see Table 2). Insert a syllable boundary before it.*

Clusters of three non-syllabic segments: schr, spr, spl, str, scl, scr, skl, skr. Clusters of two non-syllabic segments: sch, sm, sp, ps, ts, kn, sn, gn, st, dw, kw, tw, zw, th, ph, sk, sc, cl, pl, sl, bl, fl, chl, gl, kl, vl, chr, cr, pr, tr, br, dr, fr, gr, kr, vr, wr, tj (as in *zee-tje*; little see), sj, pj (as in *droom-pje*; little dream), sh, sf, fn, fj, pn

Table 2. Possible onsets (syllable-initial clusters) in Dutch.

E.g.:

kor-sten (crumbs), *amb-ten* (offices), *herf-stig* (autumnal)

There are some problems with this rule. Although *ts* is a possible syllable-initial cluster — it occurs in a number of loan words —, there is a reluctance to insert a syllable boundary in front of it in clusters of length three. E.g. *art-sen* (physicians), not *ar-tsen*; *rant-soen* (ration), not *ran-tsoen*. A similar phenomenon exists with the possible syllable-initial cluster *tw*: *ant-woord* (answer), not *an-twoord* and *Ant-werpen* (Antwerp), not *An-twerpen*. These words are interpreted as compounds, possibly for reasons of analogy or (folk) etymology.

A small-scale, informal empirical investigation showed that there is also some confusion as to where to insert a syllable boundary in clusters ending in *st*: *tek-sten* or *teks-ten* (texts), *bors-ten* or *bor-sten* (breasts), *mon-steren* or *mons-teren* (inspect). Both possibilities are equally possible. But there are also clear cases, conform to the rule: *ven-ster* (window), *bor-stel* (brush).[16]

Finally, there is the case of the cluster *str* (mostly in words of foreign origin). Following the rules, a syllable boundary can be inserted before this cluster, because it belongs to the list of possible syllable-initial clusters. However, sometimes this is clearly the wrong solution: *Cas-tro*, not *Ca-stro* and *mis-tral* not *mi-stral*. This is due to the fact that the vowel before *str* is short, and in Dutch, short vowels are normally indicated by closed syllables. The same rule contradicts the maximal syllable onset principle in the case of intervocalic non-syllabic clusters of length two (see 4.1.1). In other cases two alternatives seem equally possible: *adminis-tratie* and *admini-stratie*, *Aus-tralië* and *Au-stralië*, *mine-strone* and *mines-trone*. Individual differences in the pronunciation of these words may account for this undecidedness.

For clear cases among these exceptions, ad hoc rules should be added to the program, overriding the syllabification rules. For fuzzy cases, for which people differ in meaning, the default rules should be applied.

[16] In Trommelen (1983), *s* in onsets is analysed as extra-metrical (i.e. independent from the metrical syllable structure), and in codas it is analysed as an appendix (again independent from the syllable structure). The undecidedness in words like these is then explained by the special status of *s*, and the fact that no morphological boundary can force a decision. But no convincing independent evidence is given for the marginal status of *s* (except that it makes the rules look better).

4.1.2.2 Polymorphematic Words

Polymorphematic or complex words are words consisting of a combination of simplex words, complex words, and/or affixes. In complex words, the rules discussed earlier often lead to errors. This is due to the fact that Dutch syllabification is also guided by the principle 'divide according to morphological structure'. This principle sometimes conflicts with the basic syllabification rules. We tested a program[17], incorporating the rules discussed in the previous section, on a corpus of 5546 word forms (the letters A, W and Z in a computer-readable version of Uit den Boogaart, 1975). This program included no morphological analysis. It failed in 5.6% of the cases (707 of a total of 12647 computed boundaries) due to the morphological structure of the words. This is an unacceptable error percentage, and the experiment proves that morphological analysis is necessary to obtain high quality syllabification.

In morphological description, the different parts of a complex word are separated by morphological boundary symbols. In Dutch (as in English) two types of boundaries are considered; morpheme boundaries (+) and word boundaries (#) (Booij, 1977). Only the latter type overrules syllabification rules. E.g. compare

groen+ig groe-nig (greeny) (syllabification not overruled)
to
groen#achtig groen-ach-tig (greeny) (syllabification overruled)

Other examples of the priority of morphological structure to syllabification rules:

in#enten	in-en-ten (vaccinate)	<->	*i-nen-ten
stads#plan	stads-plan (city map)	<->	*stad-splan
zee#tje	zee-tje (little sea)	<->	*zeet-je

The third column shows the erroneous syllabification which would ensue if the syllabification rules were not overruled.

In Dutch, all prefixes and all parts of a compound are followed by a word boundary. Some suffixes are preceded by a word boundary, others by a morpheme boundary. Only word boundaries must be replaced by a syllable boundary before the regular syllabification rules apply. In our algorithm, this is done by a word form parser which splits up compounds into their parts, and by an affix analyser which indicates the boundary of all prefixes and those suffixes which are preceded by a

[17] More fully described in 5.2.3.1.

word boundary.[18] These algorithms were described in section 3.2. By means of this partial morphological analysis the problems caused by the conflicting effects of syllabification rules and the morphological principle are resolved in essence. Remaining errors are the result of deficiencies in the morphological analysis.

To minimise the probability that these erroneous morphological analyses have an adverse effect on syllabification, analysis can be limited to those cases where it is absolutely necessary. There is a category of combinations of word parts separated by an internal word boundary (*'internal words'*) which can be proved to be 'unproblematic' for syllabification. That is to say, the default application of the syllabification rules for simplex words is correct for these combinations as well. In those cases, no morphological analysis is transferred to the syllabification part of the algorithm. In our program, an analysis is only returned in the following cases (an internal word is indicated by W_i).

(i) W_i ends in a non-syllabic segment, and W_{i+1} starts with a syllabic segment (e.g., *in#enten*, to vaccinate).

(ii) W_i ends in a syllabic segment, and W_{i+1} starts with a *possible syllable-initial cluster* which is not a *cohesive cluster* (e.g., *zee#tje*, little see).

(iii) W_i ends in one or two non-syllabic segments, and W_{i+1} starts with a non-syllabic cluster, where the last segment of W_i and the first segments of W_{i+1} taken together form a *possible syllable-initial cluster* (e.g., *stads#plan*, city map).

Notice that it is not the case that *no* morphological analysis is computed in these cases (it is only after such an analysis that the restrictions can be checked). The mechanism described only diminishes the probability that an erroneous analysis is sent to the syllabification algorithm by diminishing the overall number of analyses which is sent.

It is an empirical question which suffixes in Dutch are preceded by a word boundary. The official word list claims that all suffixes beginning with a consonant and the suffixes *-aard* and *-achtig* overrule syllabification rules. But there may be a class of unclear cases. One variant of the past participle suffix (*-te*) may be a case in point. When asked to hyphenate the following words, people tend to hesitate between *s-te* (interpreting the boundary before *te* as a word boundary) and *-ste* (interpreting it

[18] Morphological analysis is only necessary when a dictionary containing information about the place of these internal word boundaries is not available and for new compounds.

as a morpheme boundary), with a slight preference for the former: *boks-te* (boxed), *fiets-te* (biked), *vors-te* (investigated), *wals-te* (waltzed), *vervlaams-te* (Flemished). It should be noted that the difference between word and morpheme boundary is not only motivated by a different effect on syllabification, but also by other phonological phenomena, e.g., the shifting of word accent and blocking effects on other phonological rules.

4.1.3 Implementation of the Algorithm

We added a subject *syllabification* to the KRS concept *word-form*. This procedure computes the syllable boundaries. Sub-problems are delegated to various other procedures (e.g. *primitive-phonemisation* and *compute-word-boundary*, attached to *word-form*; *find-syllable-boundary*, attached to *vowel-string*, and *generate-syllable-boundaries*, attached to the concept *consonant-string*. The different lists with phonotactic data (possible syllable-initial clusters, cohesive-clusters etc.) are implemented simply as Lisp lists.

4.2 A Phonemisation Algorithm[19]

Phonemisation cannot be regarded as a simple transliteration function f: S -> P, where S is a set containing the alphabet of spelling characters and P the set of phonological symbols. The relation is not isomorphic, it is context-sensitive, and involves mappings from single spelling symbols to strings of phonological symbols and from strings of spelling symbols to single phonological symbols. This creates a parsing problem when the function is applied to a string of spelling symbols.

It is quite common for a single sound to be represented by different spelling symbols (or sequences of spelling symbols) on different occasions. E.g. phoneme /a/ may be realised in spelling as <a> or <aa> depending on whether it is the nucleus of an open or a closed syllable. On the other hand, one spelling symbol may represent different sounds in different contexts. A notorious example to which we have already referred is grapheme <e>, which can be realised in speech as /ɛ/ (lax vowel), /e/ (tense vowel) or /ə/ (unarticulated vowel, *schwa*). E.g.

<perforeren> (to perforate) --> /pɛrforerən/.

[19] This section is based on parts of Daelemans 1985b.

In this case, it is a combination of stress and syllable structure information which decides how the phoneme is represented in spelling.

The parsing problem is illustrated by the transcription of <ng>, which has to be transliterated to /nγ/ in some cases (e.g. *aangeven*, to pass) and to /N/, the velar nasal[20], in other cases (e.g. *zingen*, to sing). In this case it is morphological structure (*aan#geven* versus *zing+en*) which determines the correct transcription.

Reliable phonemisation can be achieved only by means of a system of context-sensitive rules, drawing on diverse sources of linguistic information (mainly phonological and morphological, but also syntactic and semantic).

4.2.1 Overview of the Algorithm

The algorithm works in three stages: (i) syllabification and word stress assignment, (ii) application of transliteration rules, and (iii) application of phonological rules. Figure 5 provides an overview. Before applying the algorithm, a small on line exception dictionary with the pronunciation of foreign words and phonologically irregular Dutch words is checked. In a lexical database as described earlier, the phonemisation of these words and of regular words can be simply retrieved, and the algorithm would only have to be used in the case of new word forms.

(i) *Syllabification and word stress assignment.* For each input word the syllable structure and a partial morphological structure are computed or retrieved. At the same time, word stress is looked up in the lexical database.

(ii) *Transliteration rules.* Spelling syllables are transformed into phonological syllables by means of a set of mappings.

(iii) *Phonological rules.* Diverse phonological rules are applied to the output of stage (ii).

The difference between transliteration and phonology, as regards the output of these stages, roughly corresponds to the (fuzzy) traditional distinction between a broad and a narrow phonetic transcription. Sometimes there is no linguistic motivation to treat a particular rule in either (ii) or (iii). Each stage will be described in more detail in later sections.

[20] Our notation of the velar nasal is the only case in which we will deviate from the standard phonetic notation.

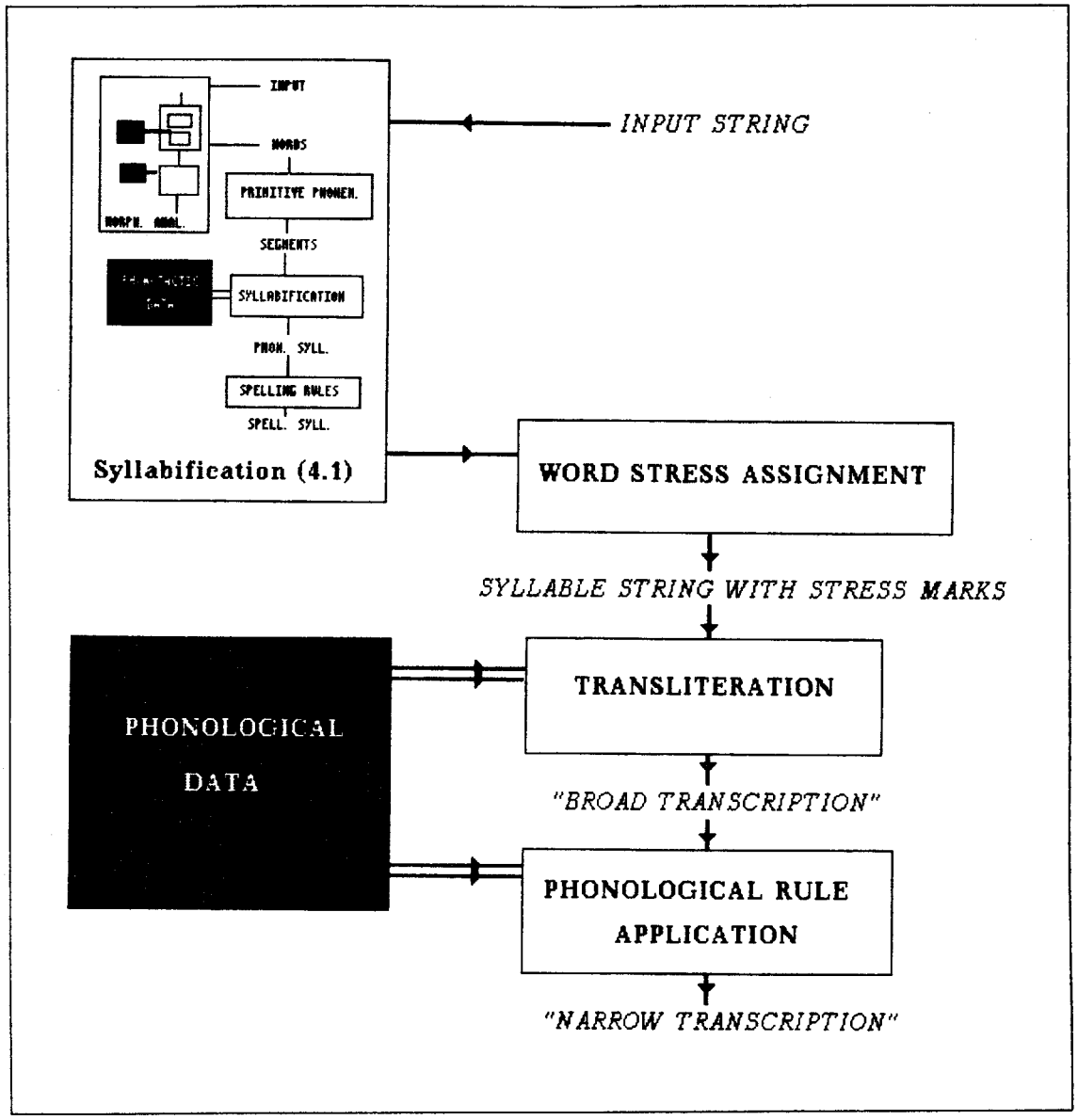


Figure 5. An outline of the phonemisation algorithm. Black boxes represent data, white boxes processes. The double arrows indicate which data are used by which processes. The single arrows indicate the input and output relations between representations and processes.

Several applications are possible for a phonemisation algorithm:

(i) The construction of a phonological representation of written text is an important step in the *automatic synthesis of speech* from printed text. An obvious application of such a system would be the construction of a reading machine for the visually handicapped (see section 8.3).

(ii) It can be used as a *tool for linguistic research* and for teaching phonology (see section 8.1).

(iii) The algorithm can also be used for the automatic generation of *pronunciation and rhyme dictionaries* or to add a level of phonological information to existing dictionaries (see section 8.2).

(iv) If the program is applied to a large enough text corpus, statistical information about the *frequency of phonemes and phoneme combinations* can be obtained. These data may be helpful in teaching Dutch as a foreign language.

Two restrictions should be kept in mind:

(i) The input text should not contain abbreviations and numbers. These must be expanded to a pronounceable form first. E.g. *21* should be transformed into *een-en-twintig*, and *bijv.* to *bijvoorbeeld* (for example). This pre-processing could be done by a *lexical analyser* (described in section 8.3).

(ii) No syntactic or semantic analysis is involved. Consequently, the *phonological phrase*, which restricts sandhi processes in Dutch, is not present as a possible domain for phonological rules, and semantic ambiguity cannot be resolved. E.g.: *ververs* /vɛrvɔrs/ (painters) versus /vɔrvɛrs/ (to refresh).

In words like these, the correct pronunciation cannot be predicted without semantic information. Our system will choose one possibility at random in these cases.

4.2.2 The Representation of Phonological Data[21]

Few textbooks agree upon an inventory and classification of Dutch phonemes. Especially the status of /ə/, /h/, /N/, semi-vowels and diphthongs is much debated, and different sets of distinctive features are proposed for categorising vowels. Classifications are closely related to the phonological rules which are used. Every theory tries to make distinctions in such a way that phonemes fall into 'natural' classes which can be treated as unities by the rules. E.g., if a set of consonants (say /l/ and /r/) always feature in the same rules in the same way, a class *liquids* can be

[21] The data presented in this section can only be used when spelling symbols have already been transformed into phonemes. I.e. they can only be used by the phonological rules, not by the transliteration rules (this distinction is made clear in the next sections). This implies that some functions must be defined twice: once applicable to graphemes and once to phonemes. The predicate vowel-p which distinguishes between vowels and consonants is a case in point.

created, containing these consonants. Our phoneme inventory is organised so as to be easily compatible with most formalisms and taxonomies currently in use in Dutch phonology.

A simple hierarchy is used (Figure 6) to describe the relations between different phonological concepts.

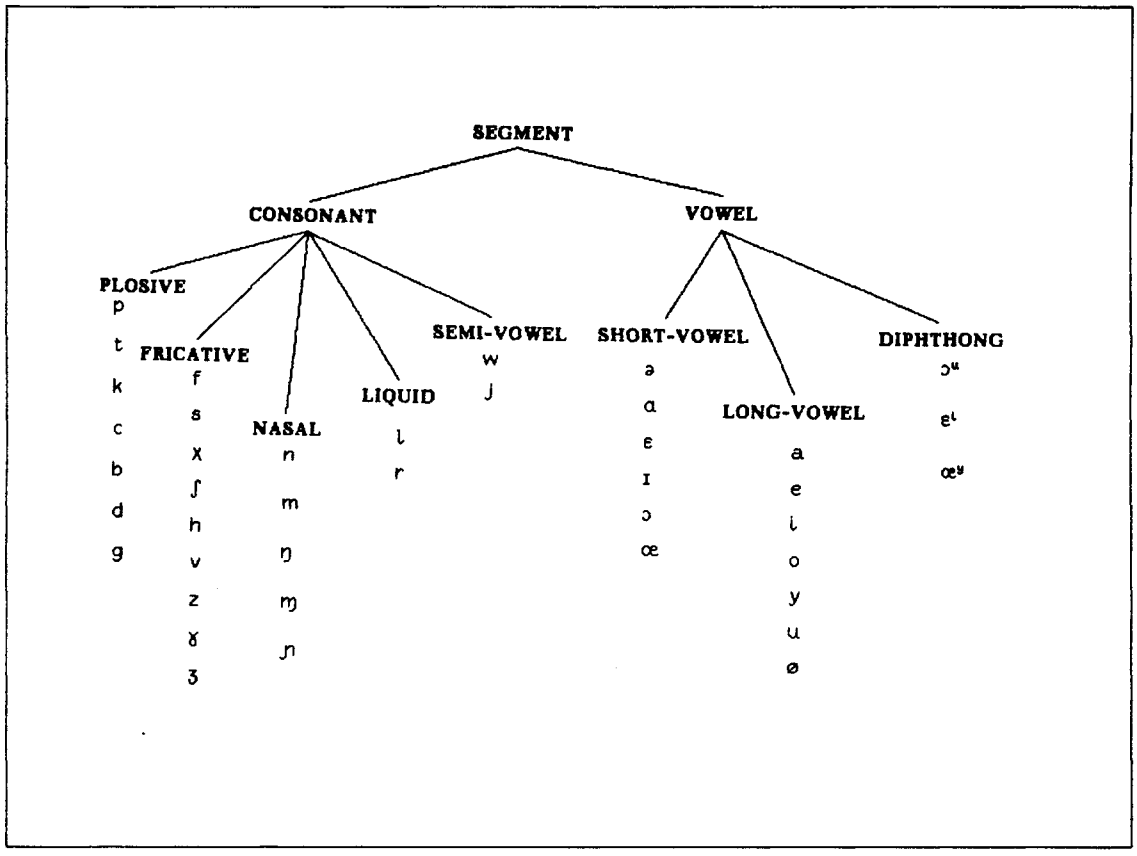


Figure 6. Hierarchy of Phonological Concepts.

The nodes of the tree refer to types. The branches denote subtype relationships between types. The phonemes listed are subtypes of the type immediately above them in the figure.

Consonants are categorised by means of the properties *manner of articulation* (plosive, fricative, nasal, ...), *place of articulation* (bilabial, alveolar, velar, ...) and *voice*. The first property is represented in the hierarchy of Figure 6, the other two are represented by means of features associated with objects (Figure 7). Vowels are categorised by means of the properties *length, manner of articulation* (rounded or not) and *place of articulation* (front, high, ...). Again, the first property is implicit in the hierarchy, and the others are represented by means of features (Figure 8).

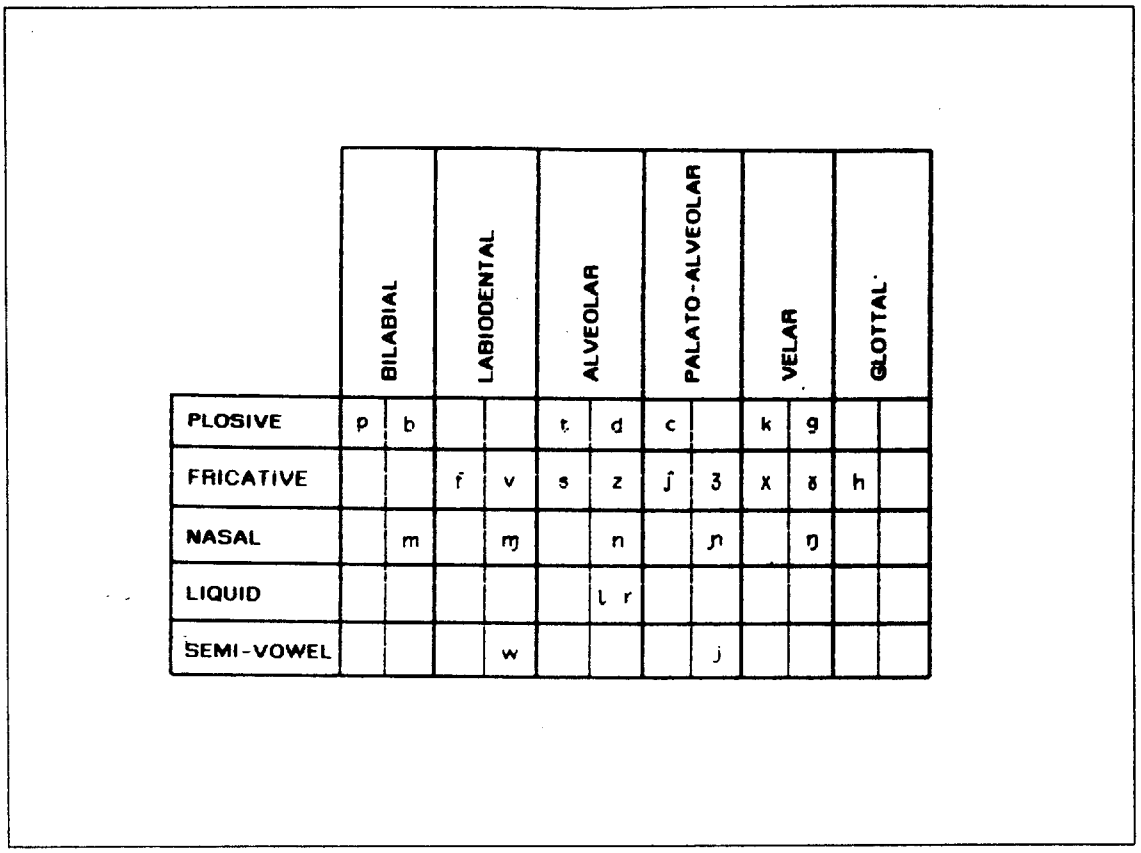


Figure 7. Features for Consonants.

The phonological knowledge base is implemented as a hierarchy of KRS concepts with associated subjects. Figure 9 shows the definition of some of the concepts in the knowledge base.

Programs can access this phonological knowledge base by means of a relatively independent interface consisting of Lisp predicates and functions in a uniform format. E.g. (obstruent-p x), (syllabic-p x), (make-short x) etc. These functions should be interpreted as questions to the lexical database: (obstruent-p x) means *'Is x an obstruent?'*. The answer can be *t* (true), *nil* (false), a numerical value (when a binary opposition is insufficient, and a gradation should be used), or a special message (when a function is not applicable to its argument). E.g. (high-p s) returns *undefined*. This is comparable to either 'irrelevant' or 'redundant' in generative phonology. Table 3 lists the access functions and transformation functions which are used in our system.

The functions can be computed in any of four ways:

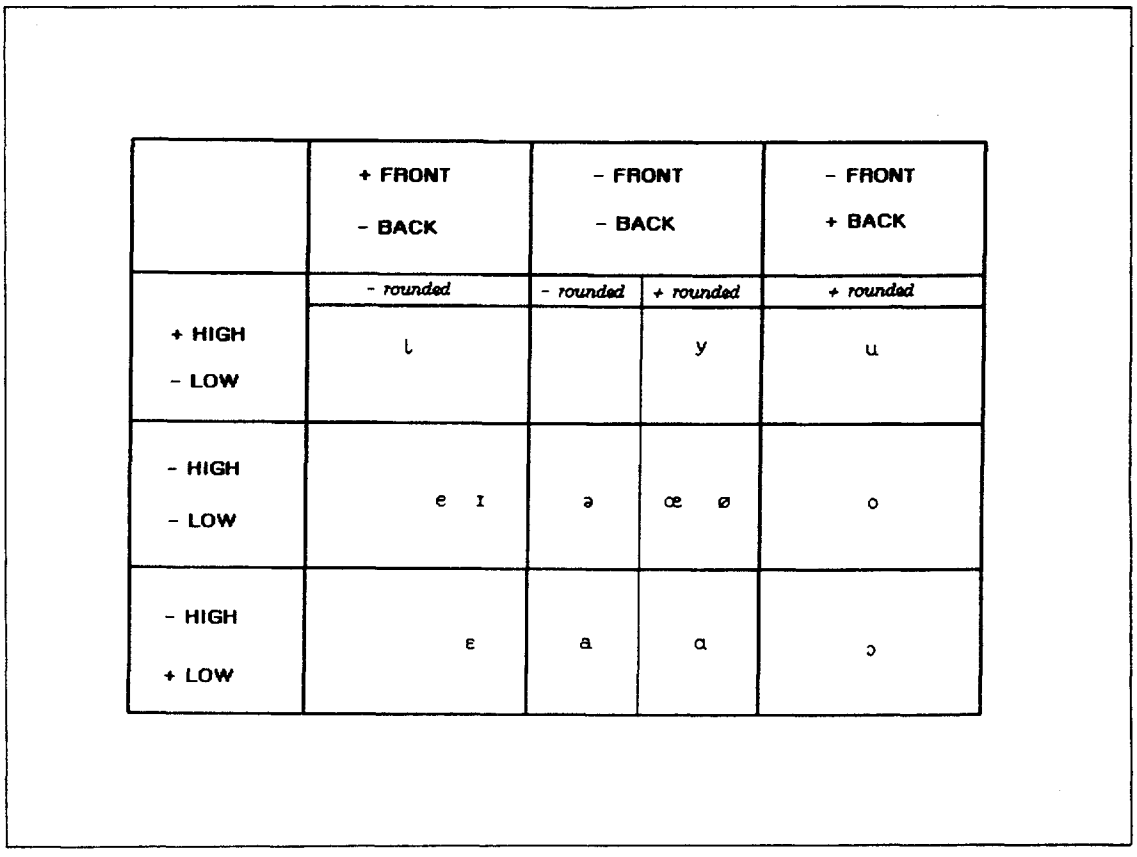


Figure 8. Features for Vowels.

(i) The information can be implicit in the hierarchy of objects. E.g., phoneme /a/ is defined as an inheritor of the object-type *long-vowel*, which inherits from the type *vowel*, which stands itself in a type/subtype relationship to *segment*. To answer the 'question' (segment-p a) the program merely needs to check the hierarchy.

(ii) The information can be stored as data associated with an object. E.g. the object /w/ (an inheritor of the type *semi-vowel*) has value 'true' for the feature roundness. To answer the question (round-p w) the program checks if a feature roundness is specified for /w/, and if so, what the value is of this feature.

(iii) The information may be inherited from objects higher in the hierarchy. E.g. the function (voiced-p a) returns true because the object /a/ is an inheritor of *long-vowel*, which is an inheritor of the *vowel*, which has value true for the feature voicedness. Note that this feature was not specified explicitly for the object /a/. In this way *default values* can be provided. E.g. the type *vowel* also has as a feature that it is stressable. All vowels inherit this feature by default. Defaults can be overruled by specifying the same feature with a different value for the

```
(DEFCONCEPT SEGMENT
  (A PHONOLOGICAL-OBJECT
     (PRINT-FORM (A STRING))))

(DEFCONCEPT CONSONANT
  (A SEGMENT
     (SYLLABIC FALSE)
     (VOICED (A BOOLEAN))
     (VELAR FALSE)                ;; This is the default
     ...
     (VOICED-VERSION (A CONSONANT))
     (DEVOICED-VERSION (A CONSONANT))))

(DEFCONCEPT NASAL
  (A CONSONANT
     (VOICED TRUE)))

(DEFCONCEPT CH22
  (A NASAL
     (PRINT-FORM [STRING "n"])
     (VELAR TRUE))                ;; Overrides default specified in
                                  ;; Concept CONSONANT
   ...
```

Figure 9. Some KRS-concepts describing knowledge about consonants.

relevant objects lower in the object hierarchy; to the object /ə/, the value nil is specified for the property stressable, although it is a vowel. The value of a feature of an object is not inherited by its inheritor if it conflicts with a known value of that feature associated with the inheritor.

(iv) Finally, the answer can be computed. The question (obstruent-p k) is answered by computing a Lisp Boolean function (or (fricative-p k) (plosive-p k)).

At present, the phonological knowledge base distinguishes 41 phonemes (some of these only occur as the result of assimilation processes) and five diacritics. We list them in Table 4. This inventory can be altered or extended at will, as can the hierarchy and the number of access functions. E.g, we could decide to give up the distinction long versus short for vowels and use tense versus lax instead. Or we could alter the hierarchy so that voiced obstruents inherit information from their voiceless counterparts. Changing or extending the knowledge base can be done by adding or modifying KRS concepts and subjects, but it can also be achieved (in a more limited way) by changing the Lisp-function interface (by renaming, adding or removing functions).

The present organisation of the phonological data derives from an eclectic choice from the opinions of various authors: Collier and Droste (1977), Collier (1981), De

LISP PHONOLOGICAL ACCESS FUNCTIONS

fricative-p (segment)	consonant-p (segment)	voiced-p (segment)
plosive-p (segment)	obstruent-p (segment)	front-p (segment)
nasal-p (segment)	nasa-liquid-p (segment)	back-p (segment)
liquid-p (segment)	bilabial-p (segment)	high-p (segment)
semi-vowel-p (segment)	palatal-p (segment)	low-p (segment)
long-p (segment)	alveolar-p (segment)	round-p (segment)
short-p (segment)	velar-p (segment)	syllabic-p (segment)
diftong-p (segment)	labiodental-p (segment)	stressable-p (segment)
vowel-p (segment)	glottal-p (segment)	

LISP TRANSFORMATION FUNCTIONS

make-voiced (segment)
make-voiceless (segment)
make-short (segment)
make-long (segment)

Table 3. Access functions to and transformation functions of phonological data.

Schutter (1978), Van den Berg (1978), Van Bakel (1976), Booij (1981) and Cohen (1961).

4.2.3 Syllabification and Word Stress Assignment

The algorithm used to compute syllable boundaries is described in section 3.3. We use a lexical database (section 3.5) and a partial morphological analysis system (section 3.2) to make possible the correct hyphenation of the whole *potential* vocabulary of Dutch; i.e. all existing words and all potential new words which may be formed by rules of compounding and affixation.[22] The syllable is central in this algorithm; it is the default domain for the application of phonological and transliteration rules. This approach deserves some explicit motivation.

[22] The necessity for the presence of morphological analysis in grapheme-to-phoneme transliteration systems has recently been argued for the case of German as well (Pounder and Kommenda, 1986).

PHONEME INVENTORY

1	@	ə	11	I	ɪ	21	l	l	31	g	g
2	y	y	12	E	ɛ	22	N	ŋ	32	k	k
3	u	u	13	A	ɑ	23	n	n	33	d	d
4	o	o	14	6	œʸ	24	m	m	34	t	t
5	0	ø	15	7	øʸ	25	G	ɣ	35	b	b
6	i	i	16	8	oᵘ	26	X	χ	36	p	p
7	e	e	17	h	h	27	v	v	37	S	ʃ
8	a	a	18	w	w	28	f	f	38	Z	ʒ
9	%	œ	19	j	j	29	z	z	39	c	c
10	O	ɔ	20	r	r	30	s	s	40	M	ɱ
									41	J	ɲ

Diacritics

42	7	øʸ
43	8	oᵘ
44	9	eⁱ
	2	ᵊ
	:	:

Table 4. Phoneme and diacritic inventory. The first column lists the concept names, the second column the internal representation, and the third column the traditional phonetic symbol. E.g., the velar nasal is a concept named CH22 and with internal representation "N".

As some important phonological rules have the syllable as their domain, it is at least necessary to represent it. Consider for example *schwa-insertion*, a rule which inserts a schwa (/ə/) between a liquid and a nasal or non-coronal consonant. This rule applies only if both consonants belong to the same syllable. E.g. compare

erg (bad) /ɛrəX/ and *melk* (milk) /mɛlək/ (schwa-insertion)

to

er-ge (bad) /ɛrγə/ and *mel-ken* (to milk) /mɛlkə/ (no schwa-insertion).

A simple pattern matching rule system would fail to make this distinction. Other rules, like *final devoicing*, a rule which devoices voiced obstruents in final position, and *progressive and regressive assimilation*, are traditionally described as applying to words or morphemes, excluding their operation at the end of syllables. However, the following examples show that it would be more correct to define them with the

syllable as their domain: e.g.

> *het-ze* (smear campaign) becomes /hɛtsə/ (progr. assimilation), *as-best* (asbestos) and *anek-dote* (anecdote) become /ɑzbɛst/ and /ɑnɛgdotə/ (regr. assimilation) and *Cam-bod-ja* becomes /kɑmbotjɑ/ (final devoicing).

Although these examples show that syllable structure is necessary, they do not prove that it is central. The centrality of the syllable is suggested by the following observations.

(i) The combination of syllable structure and information about word stress seems sufficient to transform all spelling vowels correctly into phonemes, including grapheme <e>, a traditional stumbling block in the design of Dutch phonemisation algorithms.

(ii) An implementation of our algorithm shows that most phonological rules can be defined straightforwardly in terms of syllable structure. I.e. rules which are defined in the literature with the morpheme as their domain, can be redefined with the syllable as their default domain without producing errors (rules applying at syllable boundaries also apply at morpheme boundaries).

Not everyone is convinced that it is necessary to consider the syllable as a starting point for a phonemisation algorithm. For instance, Boot (1984) claims that

> '(...) there is no linguistic motivation whatsoever for a [phonemisation WD] model in which syllabification plays a significant role (o.c. p. 63) [my translation WD].'

It is clear from the data presented here that this is not true. His own 'second generation' program uses a set of context-sensitive pattern matching rules and some form of affix-stripping. No dictionary is present. It is hard to see how his program would be able to distinguish between word-pairs such as *beven* (to shudder; no prefix) and *bevelen* (to command; prefix *be#*). They are transliterated respectively as /bevən/ and /bəvelən/. What is needed to detect affixes is morphological analysis, which presupposes a dictionary, as was pointed out in 3.3. The output of the syllabification part of the algorithm is a character string in which internal and external word boundaries (# and ##) and syllable boundaries (=) are inserted.

Parallel with morphological analysis, the primary word stress of each word and of each part of a compound is looked up in the dictionary which is used for morphological analysis. This information is indicated by an asterisk (*) before the stressed syllable. Word stress can be *computed* instead of looked up for compounds.

Some examples of the output of this stage:
vergelijking (comparison)
*##ver#ge#*lij=king##*
een heerlijke appelmoes (a delicious apple-sauce)
*##een##*heer=lij=ke##*ap=pel#*moes##*
herfststorm (autumnal tempest)
*##*herfst#*storm##*

4.2.4 Processing Syllable Strings

Before discussing the transliteration and phonological rules in more detail, we return once more to the object-oriented organisation of the phonological data. Every syllable in the hyphenated input string becomes an instance of the type *syllable* (implemented as a KRS concept), which has a number of features (Figure 10 lists these features, and their value for one particular syllable).

OBJECT-TYPE	FEATURES	EXAMPLE
		THE SYLLABLE KING IN ##VER#GE#LIJ-KING## (COMPARISON)
SYLLABLE	SPELLING	KING
	IS-CLOSED?	TRUE
	IS-STRESSED?	FALSE
	PREVIOUS-SYLLABLE	SYL003463
	NEXT-SYLLABLE	FALSE
	EXTERNAL-WORD-BOUNDARY-ON-RIGHT?	TRUE
	INTERNAL-WORD-BOUNDARY-ON-RIGHT?	FALSE
	STRUCTURE	"K" "I" "NG"
	ONSET	/k/
	NUCLEUS	/ɪ/
	CODA	/ŋ/
	RHYME	/ɪŋ/
	TRANSCRIPTION	/kɪŋ/

Figure 10. Features of the object SYLLABLE and an example.

The value of some of these can be set by means of the information in the input:

spelling, *is-closed?* (true if the syllable ends in a consonant), *is-stressed?* (true if the syllable carries word stress), *previous-syllable* (a pointer to the particular syllable immediately preceding the one being processed), *next-syllable* (a pointer to the next syllable), *external-word-boundary-on-right?* (can be true or false), and *internal-word-boundary-on-right?* (can be true or false) are initialised this way. The names of these features are self-explanatory. Their values are used by the transliteration and phonological rules. For other features, the value must be computed: *structure* is computed by means of the value for the *spelling* feature and the predicate vowel-p, which distinguishes spelling vowels from spelling consonants. The value for this feature reflects the internal structure of the spelling syllable: *nucleus* (the vocalic element of a syllable), *onset* (the consonantal part optionally preceding the nucleus) and *coda* (the consonantal part optionally following the nucleus). E.g. the structure of the spelling syllable *herfst* is "h" "e" "rfst". As the syllabification part of our algorithm incorporates a similar segmentation (primitive phonemisation), the structure feature can be provided a value without computation as well.

The values of the features *onset*, *nucleus*, and *coda* (this time of the phonological syllable) are computed by means of the transliteration and phonological rules. Their initial values represent the spelling, and their final values the pronunciation. The rules take the values of these features as their input and can change them. The *rhyme* of a syllable is nucleus and coda taken together, and the feature *transcription* stands for the concatenation of the final (or intermediate) values of onset, nucleus and coda. A useful metaphor is to envision the *onset*, *nucleus* and *coda* features as pieces of paper. At first, the spelling is written down on them. Later, the rules may wipe out what is written down, and write down other symbols. When all rules have been applied, the pieces of paper can be pushed against one another and the result is the phonological transcription of the spelling syllable.

Apart from the information specified by features of the object and the phonological data discussed earlier (all accessible in a uniform Lisp function format), the rules use primitive string-manipulation functions to check their conditions and to change the values of the features. If the transformation from spelling to phonological syllable has been achieved for all syllables in the input, their transcriptions are concatenated, and the result is returned.

4.2.5 Transliteration Rules

An input string is scanned twice from left to right: by the transliteration rules and by the phonological rules.[23] Transliteration rules are mappings of elements of syllable structure to their phonological counterpart. E.g. the syllable onset <sch> is mapped to /sX/, nucleus <ie> to /i/, and coda <x> to /ks/. Conditions can be added to make the transliteration context-sensitive: the syllable onset <c> is mapped to /s/ if a front vowel follows and to /k/ if a back vowel follows. Here the distinction between transliteration and phonological rules becomes fuzzy. E.g.

cola /kolɑ/ versus *cent* /sɛnt/

We have already mentioned that syllable structure and word stress information suffice to disambiguate spelling vowels. E.g. grapheme <e> when occurring in a stressed and open syllable becomes /e/, in an unstressed syllable /ə/ (schwa), and in a stressed and closed syllable it becomes /ɛ/. Some examples (external word boundaries are omitted for clarity's sake):

**cent* (closed and stressed) becomes /sɛnt/,
**ne=ro* (open and stressed) becomes /nero/,
**lo=pen* (to run; closed and unstressed) becomes /lopən/, and
**ro=de* (red; open and unstressed) becomes /rodə/.

Other spelling vowels are transformed in a similar way.

Apart from this mapping, some modifications are necessary to transform spelling syllables into phonological syllables: e.g. the cluster <ng>, sometimes split into two parts during syllabification, must be rewritten as one phoneme: **zin=gen* (to sing) becomes */ziNən/*. There are about forty transliteration mappings and modifications.

4.2.6 Phonological Rules

The phonological rules apply to the output of the transliteration mappings. They are sequentially ordered, but alternative orderings, triggered by arbitrary properties of the input, can coexist with the default order. This default order can be changed. Each rule is a subtype of the *phonological-rule* concept, which, like any rule, has five

[23] This is by no means an essential property of the algorithm, transliteration and phonological rule application could easily be done in one pass. The present organisation is only intended to reflect a conceptual (theoretical) distinction between both types of rules.

features: active-p, application, conditions, actions and examples. An example of the KRS-implementation of a rule is given in Figure 11.

```
(DEFCONCEPT SCHWA-INSERTION
  (A PHONOLOGICAL-RULE
     (ACTIVE-P TRUE)                ;; The rule is turned on

     (APPLICATION                   ;; Actually, this subject is inherited
       (DEFINITION                  ;; from the general concept Rule
         [IF (>> CONDITIONS) (>> ACTIONS)]))

     (CONDITIONS                    ;; coda-first and coda-second refer
       (DEFINITION                    ;; to the first and second character
         [FORM (AND (LIQUID-P CODA-FIRST) ;; of the coda
                    (ALVEOLAR-P CODA-FIRST)
                    (OR (BILABIAL-P CODA-SECOND)
                        (LABIODENTAL-P CODA-SECOND)
                        (VELAR-P CODA-SECOND)))]))

     (ACTIONS
       (DEFINITION
         [FORM (CHANGE-CODA
                  (STRING-APPEND CODA-FIRST
                                 (>> PRINT-FORM OF SCHWA)
                                 CODA-SECOND))])))
```

Figure 11. A simplified KRS-concept for the schwa-insertion rule.

Active-p can be true or false. If it is set to true, the rule can be executed. This way, the effect of the 'deletion' or the 'addition' of a particular rule can be studied. If a rule is active, first the conditions specified in the *application* part of the rule are checked. This is a Lisp expression which evaluates to true or false. Conditions mostly consist of checking for phonemes with specific properties in the onset, nucleus or coda of a syllable, and inspecting the context of the syllable within the specified domain. If the result is true, the action part of the application feature of the rule is executed. Actions consist of modifications of the values for the features onset, nucleus and coda of a particular syllable or its neighbour (changing what is written on the 'pieces of paper'). Actions may also involve the triggering of phonological processes. E.g. in the case of schwa-insertion, a complete phonological process (*re-syllabification*) should be triggered. E.g.:

*ver=*berg* (to hide) -> (schwa-insertion) -> /vərbɛrəX/ -> (re-syllabification) -> /vər=bɛ=rəX/

Intermediate values for the features are unrecoverably lost (the output of one rule serves as the input of the next), unless they are explicitly stored at some specific time intervals. E.g. we could copy what is written on the pieces of paper each time a rule has been applied. This way a detailed derivation is obtained.

If a rule was activated, the conditions satisfied, and the actions undertaken, two 'book-keeping' activities occur:

(i) The name of the rule is entered in the *rule history* associated with each input string. This way the derivation of an input string can be traced.

(ii) The input-string to which the rule applied is stored in the feature *examples* of this rule. This way, interesting data about the operation of a rule can be obtained.

Some important rules included in the program are listed in Appendix A.8 with a short description, accompanied by a few examples (computed by the program). A trace of the derivation was added for each example. The order in which the rules appear in this derivation for each syllable reflects the order in which they were applied to that syllable. The numerical value preceding the names of the rules in the derivation reflects the overall temporal order in which the rules were activated by the program.

4.2.7 Evaluation of the Program

In Appendix A.4, the result of applying the program to a randomly chosen Dutch text is demonstrated. The input text (A.4.1), the output of the intermediary syllabification and word stress retrieval stage (A.4.2), the southern Dutch transcription (Flemish, A.4.3) and the northern Dutch transcription(A.4.4)[24] are presented. Figure 12 gives short fragments of these different representations. The overall results were satisfying: of 475 predicted syllable boundaries, 6 turned out wrong (success-rate 99.99%), of 3639 phonemes produced in the output, 27 were judged wrong by an independent linguist (again a success-rate of 99.99%), and finally, 23 out of 964 phonemised word tokens were rejected (success-rate 99.98%). The mistakes made by the program can be reduced to four categories:

(i) *Ambiguity*: The spelling form <een> can be either /ən/ (a/an, determiner) or /en/ (one, numeral). Usually, the difference is reflected in spelling (the numeral is spelled <één>), but this is not obligatory. With syntactic information, this

[24] We have implemented (our impression of) a simplification of the two major variants of standard (educated) Dutch: southern and northern Dutch. The former can be derived from the latter by deleting a few rules (initial devoicing and vowel diphthongisation), and by some additional modifications. In practice, aspects of both the southern and northern dialect may be combined in a single speaker.

Hij bewoont de kamer naast de mijne.
De volgende dag belde hij mij.
Je kamer staat er bij
zoals je hem hebt verlaten.
Ik durfde niet te vragen
naar de toestand van de meubels.

(+hij == be +woont == de == +ka mer == +naast == de == +mij ne)
(de == +vol gen de == +dag == +bel de == +hij == +mij)
(je == +ka mer == +staat == +er = +bij)
(+zo = +als == je == +hem == +hebt == ver +la ten)
(+ik == +durf de == +niet == te == +vra gen)
(+naar == de == +toe = +stand == +van == de == +meu bels)

ˈhɛˑ bəˈwoᵘn də ˈkaməɾ ˈnaz də ˈmɛˑnə
də ˈfɔlɣəndə ˈdɑɣ ˈbɛldə ˈhɛˑ ˈmɛˑ
jə ˈkaməɾ ˈstat ˈɛrˌbɛˑ ˈsoᵘˌwɑls jə ˈhɛm ˈhɛpt fərˈlatə
ˈɪg ˈdœɾəvdə ˈnɪ tə ˈfraɣə
ˈnaɾ də ˈtuˌstɑnt ˈfɑn də ˈmøʸbəls

ˈhɛˑ bəˈwɔn də ˈkaməɾ ˈnaz də ˈmɛˑnə
də ˈvɔlɣəndə ˈdɑɣ ˈbɛldə ˈhɛˑ ˈmɛˑ
jə ˈkaməɾ ˈstat ˈɛrˌbɛˑ ˈzoˌwɑls jə ˈhɛm ˈhɛpt fərˈlatə
ˈɪg ˈdœɾəvdə ˈnɪ tə ˈvraɣə
ˈnaɾ də ˈtuˌstɑnt ˈfɑn də ˈmøbəls

Figure 12. Fragments of input text, intermediary representation, southern Dutch transcription and Northern Dutch transcription of a random text. Taken from Appendix A.4.

ambiguity can be resolved when the numeral is used as the head of an NP. Unfortunately, even then, cases remain which cannot be resolved (compare e.g. *één boek*; one book and *een boek*; a book).

(ii) *Foreign words and names*: The majority of errors is of this kind: from French <beige>, <college> and <plafond>; from English <More>, <tower>, and <Charterhouse>; and from German <Holbein>. These words must be added to the exception dictionary which is consulted before the application of the rules. Note that the transliteration coincides with the pronunciation of these words by Dutch speakers not familiar with the foreign language.

(iii) *Syllabification, morphological analysis and stress assignment*: The word <glorieuze> (glorious) was hyphenated glo-rieu-ze (parallel with nieu-we, new) instead of glo-ri-eu-ze. This shortcoming of the syllabification algorithm was corrected. The parser understandably failed to see the compound structure of <Luxaflex> (a brand name). Accordingly, no stress was assigned to the syllable *flex*, which was therefore transcribed /flðks/ instead of /flɛks/. A general rule seems to exist which forbids a schwa before an *x* (compare *exámen*, examination, which would lead to an incorrect phonemisation *with* correct stress-information as well as without). Incorrect morphological analysis explains the faulty phonemisation (/bɛinɑ/ instead of /bɛina/) of <bijna> (almost). In five other cases, morphological analysis generated a syllabification error; we give the incorrect hyphenation: *mee-stal, voo-roor-log-se, moois-te, waa-rach-ter,* and *ver-bij-sterd*. The correct hyphenations are *meest-al, voor-oor-log-se, mooi-ste, waar-ach-ter* and *ver-bijs-terd*. However, these errors did not result in incorrect phonemisation.

(iv) *Rules*: In two cases, the definition of the phonological rules generated an error. In <melancholie>, *intervocalic-voicing* voiced the /x/. In <centimeter>, *plosive-to-fricative* transformed <t> into /s/, because it interpreted *ti* as a variant of the suffix *-tie*. These rules could be improved by adding new conditions.

4.3 Conclusion

The object-oriented approach which we adopted in the description of some morphological data and processes in Chapter 3 was extended to phonological data and two phonological processes: syllabification and phonemisation. *Phonological data* are represented by means of generalisation hierarchies and features of objects in these hierarchies (subjects of concepts in KRS terminology). Several dependency relations between processes and data were emphasised: *syllabification* relies on partial morphological analysis, lists of language-specific phonotactic data (possible clusters), and a (more or less) universal phonological principle (the maximal syllable onset principle). *Phonemisation* needs syllabification, stress assignment (or retrieval) and a number of transliteration and phonological rules. Both the syllabification and the phonemisation algorithms reached a nearly-perfect accuracy when tested on a random Dutch text.

PART III

APPLICATIONS

The number of ways in which a modular and extensible model such as the one described in Part II can be applied is immense, and covers the complete domain of language technology.

In Chapter 5, the Author Environment — a software tool for authors — is introduced, and two of its modules: automatic hyphenation (Ch. 5) and automatic detection and correction of spelling and typing errors (Ch. 6) are discussed in detail. The programs described in these chapters present a solution to problems generated by the particular way in which Dutch compounds are formed. Where gaps in our linguistic knowledge base prevent pricipled solutions to some sub-problem, or to improve efficiency, heuristics are proposed.

A rapidly growing field in Artificial Intelligence is ICAI (Intelligent Computer Assisted Instruction). An ICAI system contains — apart from knowledge about the subject-matter to be taught — a model of the student, diagnosis heuristics, an explanation module and educational strategies. In Chapter 7, a prototype ICAI system for the teaching of one aspect of Dutch morphology (verbal inflections) is described.

Rule testing devices offer a radically new way of doing linguistics. They accelerate the development of rule systems and theories, and provide powerful ways of controlling complex rule interactions and side-effects. A rule testing device for Dutch phonology is demonstrated in section 8.1.

In section 8.2, the potential use of linguistic algorithms in lexicography is exemplified by means of a Flexible Dictionary System which allows the semi-automatic creation, extension and updating of lexical knowledge bases.

A morpho-phonological model is indispensible as a module in larger natural language processing systems. A morphological component is needed in machine translation and dialogue systems as a part of the syntactic generation and interpretation programs. A phonological stage is necessary to make text-to-speech systems (reading machines)

and speech understanding systems possible. These systems are an essential part in any application incorporating a speech interface. The transportability and modularity of object-oriented systems makes them ideally suited for integration into larger systems. In section 8.3, the role of our phonemisation algorithm in a speech synthesis system will be discussed, and the concept of a lexical analyser (a general purpose front-end to natural language interpreting systems) will be introduced.

CHAPTER 5

Automatic Hyphenation in an Author Environment

5.1 The Author Environment

An Author Environment (AE)[1] is a set of interacting computer programs making up an intelligent software tool which helps authors and editors in all activities related to writing. Its aim is to make writing easier by allowing the author to devote his attention to content rather than to form.

An ideal AE should assist in all stages of the writing process which are traditionally distinguished (e.g. Bartlett, 1982; Hayes and Flower, 1980; Smith, 1982): in the *planning stage* by providing the relevant information (e.g. bibliographic databases, text retrieval systems etc.) and by helping to set up goals; in the *writing stage* by helping in text and paragraph structuring and word selection; and in the *reviewing stage* by finding errors, suggesting corrections and improvements, and decreasing the complexity of editing. We will be concerned exclusively with the writing and reviewing stages.

An AE differs from current word processors in that linguistic knowledge is integrated into it, which extends the possibilities of the user considerably. The idea is

[1] A system incorporating some of the ideas described in this section is currently being developed at the university of Nijmegen by Prof. G. Kempen and his co-workers in the framework of an ESPRIT project of the EC. See Kempen, Anbeek, Desain, Konst, De Smedt, 1986 for a recent overview, and also Konst, Daelemans and Kempen, 1984 and Daelemans, Kempen and Konst, 1985.

to keep several linguistic representations of the text available. Apart from the normal *orthographic* representation (alphanumeric characters, punctuation) which we find in present day word processors, several linguistic representations and an on-line dictionary are available as well. The representations supported by an AE give access to the *grammatical structure* of the sentences (produced by a syntactic parser), the *morphological structure* of words (produced by a morphological parser) and the *syllable structure* of words (produced by a syllabification algorithm). These representations serve in a multitude of useful applications.

At the word level we can hyphenate words while formatting a text, suggest synonyms, check the spelling of words, analyse the structure of a word, give the pronunciation of a word, or compute a particular form of a word.

At the sentence level, applications include checking grammar and punctuation, drawing attention to ambiguous sentences, and a number of linguistic transformations for which the propagation of features is needed. For example, if we want to pluralise a noun, we also have to modify determiners, pronouns and finite verbs that are dependent on it. Linguistic transformations which seem useful are the aforementioned singular/plural transformation, active/passive, declarative/interrogative, main clause/subordinate clause (this involves a change in word order in Dutch) and male/female. All editing related to a particular transformation should be done automatically (in spreadsheet fashion), using the underlying linguistic representations. In many languages such a facility is expected to be time saving.

At the text level, too, some of these transformations can be applied, and text can be checked for consistency in the use of a particular spelling system. Another useful facility is a readability score. Most existing scores are based on quantitative judgements (frequency measures, e.g. Flesch, 1976). The different linguistic representations would make additional qualitative judgments possible. Furthermore, applications can be envisioned in which AE helps in the structuring of a text, or even in its creation (report generation). Finally, common word processing functions used in editing such as *delete word*, *character backward* etc. could be broadened to linguistic functions such as *jump to subject*, *delete syllable*, *transpose constituents* and others.

All these facilities should not overwhelm the user of AE; a transparent user interface which avoids linguistic jargon, and which is easy and pleasant to use should be developed. Intuitively, a mouse, menu and window system with icons seems preferable, but more research is needed to compare the merits of different

approaches. A *mouse* is a little hand-held box with one or several buttons on it, and connected to a computer terminal. It can be rolled on a desk causing the cursor to move on the screen in a fashion analogical to the movement of the mouse on the desk. Most mouse systems allow the user to indicate, 'pick up' and 'move' things on the screen by *clicking* on them (position cursor on it and click the mouse button). A *menu* is a list of options between which a user can choose at some point in the execution of a program. Users choose by clicking with the mouse on the selected option. A *window system* is a program that allows users to have a video screen which is divided into several areas (windows), each of which is connected to a different process. E.g. they can write a text in a text-editor window, inspect the output of a style verification program in another window, and browse through a text in still another text-editor window. All these windows can be visible at the same time. Windows can be moved, reshaped, created and deleted at will by the user. *Icons* are drawings representing concepts. E.g. users can delete a file containing a draft of their article by moving a picture representing this file to the picture of a dustbin on the screen. The interface will also contain facilities to manipulate linguistic trees (representing the grammatical structure of sentences) directly (see Desain, 1986).

As described here, AE is a fairly revolutionary system. It is scheduled to be operational (at least in a prototype form) by 1988. Systems which offer part of this functionality already exist for English, e.g. IBM's EPISTLE project (also called CRITIQUE; Miller, Heidorn and Jensen, 1981) comments on spelling, grammar and style. The UNIX Writer's Workbench (Cherry, 1981) offers readability measures and an abstractness measure, and checks for punctuation, word use and spelling. However, for Dutch a different approach is often needed, as will become clear in the next sections and in the following chapter. We will describe two parts of the AE which directly rely on the linguistic model outlined in Part II: automatic hyphenation (this chapter) and error detection (Chapter 6).

5.2 Automatic Hyphenation[2]

Computers are notoriously bad at hyphenation. When the type-setting of newspapers began to be fully automated, jokes about

[2] This section is based in part on Daelemans (1985c).

'the-rapists who pre-ached on wee-knights' soon began to circulate.
D. E. Knuth

5.2.1 Background

A hyphenation algorithm incorporates a set of rules to find the positions in a word where a syllable boundary can be inserted. Hyphenation programs are used in type-setting and word-processing environments, most often as part of a text-formatting system.

In order to provide for the possibility of having justified paragraphs (paragraphs with straight margins on both sides), it is necessary to be able to split long words. The *justification process* normally works in two stages: filling and adjusting. A line is *filled* by inserting words in it until the current-line-length (a global variable) is reached. If at that point a word cannot be fitted on the line, an attempt is made to hyphenate it. Afterwards, spaces between words are increased to spread out the words in the line to the current-line-length (this is called *adjusting*). Figure 1 shows the effect of filling, adjusting, and hyphenating, respectively.

If no hyphenation is possible, justification often leads to ugly 'white holes' within lines, especially in languages like Dutch and German, where long words are the rule rather than the exception. Dutch presents some special hyphenation problems, as was pointed out in section 4.1; most notably the influence of some word-internal morphological boundaries on hyphenation makes the construction of a fool-proof hyphenation algorithm a stingy problem.

5.2.2 Adapting the Syllabification Algorithm

A principled solution to the problem is available by means of a slight adaptation of the syllabification algorithm described in section 4.1. The fact that this algorithm takes spelling as input is convenient for a hyphenation application because this restricts the adaptation to the introduction of a number of spelling and stylistic rules[3]. When used in a hyphenation program, these rules are added to the algorithm as a

[3] The rules we use are based on the rules for Dutch hyphenation listed in the official *Woordenlijst van de Nederlandse Taal* (Word List of the Dutch Language, 1954) on the one hand, and on typographical practice on the other hand.

(i)

Algorithms and heuristics for natural language processing
should be founded
on a computational theory which is (psycho-)linguistically motivated. Two
generations of hyphenation programs, one based on list search, the other
on pattern matching, have failed in this respect and have therefore not
succeeded in offering a general solution to the problem. An alternative
approach which relies on a theory about the interaction between
morphology and phonology is proposed.

(ii)

Algorithms and heuristics for natural language processing should be founded on
a computational theory which is (psycho-)linguistically motivated. Two
generations of hyphenation programs, one based on list search, the other on
pattern matching, have failed in this respect and have therefore not succeeded
in offering a general solution to the problem. An alternative approach which
relies on a theory about the interaction between morphology and phonology is
proposed.

(iii)

Algorithms and heuristics for natural language processing should be founded on
a computational theory which is (psycho-)linguistically motivated. Two
generations of hyphenation programs, one based on list search, the other on
pattern matching, have failed in this respect and have therefore not succeeded
in offering a general solution to the problem. An alternative approach which
relies on a theory about the interaction between morphology and phonology is
proposed.

(iv)

Algorithms and heuristics for natural language processing should be founded on
a computational theory which is (psycho-)linguistically motivated. Two genera-
tions of hyphenation programs, one based on list search, the other on pattern
matching, have failed in this respect and have therefore not succeeded in
offering a general solution to the problem. An alternative approach which re-
lies on a theory about the interaction between morphology and phonology is pro-
posed.

Figure 1. Four versions of the same paragraph. As it was entered in the text editor (i), filling only (ii), filling and adjusting (iii), filling, adjusting and hyphenation (iv). The formatting system used was UNIX troff.

post-processing module. The rules included in the program are the following:

(1) It is not allowed to insert a syllable boundary before an *x:* E.g. *exa-men*, not *e-xa-men* (examination).

(2) The string *oo* before *ch* is reduced to a single *o:* e.g. *goochelaar* (magician) is hyphenated *go-che-laar*.

(3) A double vowel is reduced to a single vowel before the diminutive suffix *#tje*, if the singular non-diminutive form of that word is spelled with a single vowel. Thus, *autootje* (little car) is split as *au-to-tje*, and *vlaatje* (little pie) is split as *vla-tje*. A lot of confusion exists about this aspect of Dutch spelling. Sometimes we find the form *auto'tje* (little car), analogous to *auto's* (cars), sometimes *autoos* (cars), analogous to *autootje*. Only *autootje* and *auto's* are officially correct.

(4) If possible, word forms should be split in the middle. Peripheral split-ups are considered ugly. E.g. *ato-mair* (atomic), not *a-to-mair*, and *bur-ge-meester*, not *bur-ge-mees-ter* (mayor).

(5) Internal word boundaries are a good place to split up a word (*on-interessant, kwart-slag*), even when this contradicts rule (4). E.g. *a-gnostisch* (agnostic). Boundaries immediately to the left or right of such a boundary are not very well suited (*onin-teressant, massage-heugen*) because they tend to make identification of the different word parts more difficult.

(6) Splitting up short words (less than five spelling characters long) produces an ugly effect. E.g. *la-de* (drawer).

In the current implementation of the syllabification algorithm of section 4.1, these constraints are implemented by means of a filter which removes some hyphens in the output of the syllabification algorithm on the basis of the aforementioned rules. A more efficient approach would be to *prevent* the insertion of these boundaries in the first place, but this would obfuscate to some extent the modularity of the phonological and the spelling parts which now exists.

The approach described here to the hyphenation of Dutch words is infallible in principle; remaining errors are due to shortcomings of the morphological decomposition algorithm, incompleteness of the lexical database (see Chapter 3) or semantic ambiguity, but it is also a computationally expensive approach. In the next section we will show how phonotactic restrictions can be applied to make feasible high-quality hyphenation with a less expensive algorithm.

5.2.3 Phonotactic Restrictions

In some cases, a syllable boundary can be predicted with full certainty, even when no morphological analysis is performed. This is the case whenever *phonotactic restrictions* exclude all but one possibility.

In the context of hyphenation, we define phonotactic restrictions as *'restrictions on the form and length of onset, nucleus and coda, and on the combination (co-occurrence) of nucleus and coda'* (see section 4.1.1). E.g. take the word *post#zegel* (stamp). Syllabification rules must split the cluster *stz*. There are four possible places for the hyphen: *-stz, s-tz, st-z* and *stz-*. The clusters *stz* and *tz* do not belong to the set of possible onsets for Dutch, and *stz* does not belong to the set of possible

codas. Only one possibility remains: *st-z* (*post-zegel*). In this case morphological analysis, which precedes syllabification, would have operated vacuously.

Due to the restrictions on cluster length, the number of hypotheses about the place of the hyphen which must be checked never exceeds four. E.g., a cluster consisting of five non-syllabic segments (each segment represented here by C) has only three possible syllable boundaries: CC-CCC, CCC-CC, and CCCC-C, because a coda can never be more than four segments long, and an onset never more than three.

When the restrictions leave open more than one possibility, morphological analysis is necessary to determine the syllable boundary with quasi-certainty. E.g. *ont#springen* (to rise): possible syllable boundaries are *nt-spr*, *nts-pr*, and *ntsp-r*. Only the last case can be excluded because of the impossible cluster *ntsp*. Morphological analysis must decide between the two other possibilities. In this case, the syllabification rules would choose *nt-spr*, which is correct, but only accidentally so; compare *abonnements-prijs* (subscription fee).

If diaeresis and hyphen are used properly, syllable boundaries in vowel clusters can be predicted with absolute certainty, on the basis of restrictions on vowel combinations. E.g., if a cluster of spelling vowels can be reduced to two syllabic segments, the syllable boundary is always located between them. Therefore, *beaam* (agree) is hyphenated *e-aa*. Again, no morphological analysis is necessary to find this syllable boundary. A deterministic vowel string parser was described in section 4.1.

In section 5.2.3.1, a program will be described which uses phonotactic restrictions to find syllable boundaries. We will also present some statistical data obtained by using it on a corpus of Dutch word forms and a performance evaluation. This program can be used profitably in hyphenation systems (1) to constrain the number of dictionary lookups during morphological analysis (this possibility is discussed in section 5.2.3.2) and (2) as a basis for an autonomous hyphenation system without morphological analysis in some applications: notably (a) a *partial hyphenation* to assist justification algorithms, and (b) in a user-friendly *interactive hyphenation* environment. These possibilities are described in section 5.2.3.3.

5.2.3.1 CHYP, a Cautious HYphenation Program

CHYP is a program which refuses to insert a syllable boundary whenever it does not have absolute certainty that there is only one possible solution. E.g. in *postzegelverzameling* (stamp collection) the program would insert a syllable boundary three times *post-zegel-ver-zameling*. Three other boundaries (*ze-gel*, *za-me-ling*) cannot be predicted with certainty.

Description of the Program. Figure 2 gives a survey of the program and an example.

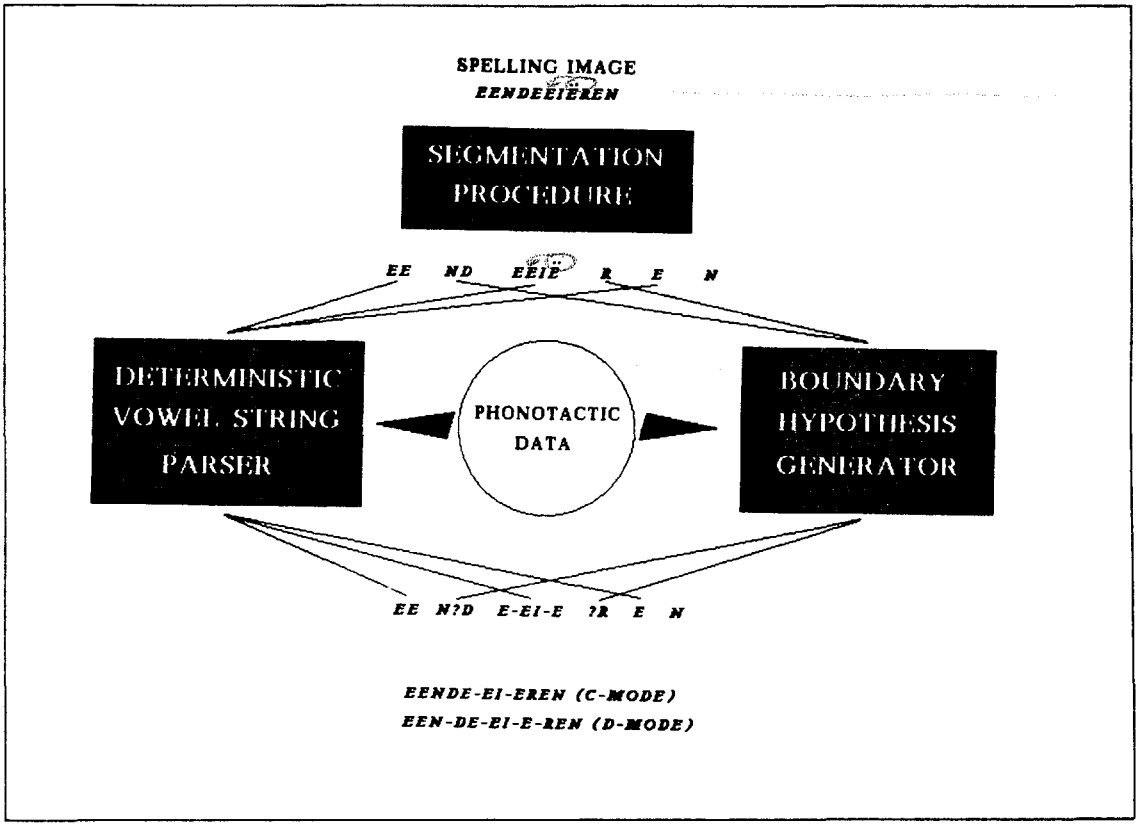


Figure 2. Overview of CHYP and an example (*eendeeiëren*, duck eggs)

The input spelling image of a word form is segmented into a head, a middle and a tail. *Head* and *tail* (word-initial and/or final consonant clusters) are ignored by the program because they need not be split. In the *middle* part, vowel strings and consonant strings are separated. Strings of spelling vowels are transferred to the *Deterministic Vowel string Parser* (DVP), discussed in section 4.1, which inserts syllable boundaries at the appropriate places.

Consonant clusters are handled by the *Boundary Hypothesis Generator* (BHG). This submodule generates and tests hypotheses about syllable boundaries. For instance in splitting a cluster *dr*, the following hypotheses are generated: *-dr, d-r, dr-*. They are tested by checking whether the part left of the boundary is a possible syllable coda, and the part right of it a possible syllable onset. If this is the case, the hypothesis is accepted and pushed on a stack (this is the case for *-dr* and *d-r*); if not, it is rejected (*dr-*). Note that hypotheses are generated by inserting tentative boundaries from left to right. This implies that the first hypothesis accepted is mostly equal to the boundary which would be suggested by applying the syllabification rules in section 4.1 (*Maximal Onset Principle*: 'take the longest possible right part'). An exception to this is the treatment of clusters of length two. In that case, the cluster is treated as a *cohesive cluster* instead of being split in the middle. E.g., the first hypothesis for *st* in *pastei* (paste) is *pa-stei*, and not *pas-tei*. A simple test can be (and has been) incorporated into the algorithm to solve this problem, so that the hypothesis which is pushed first on the stack always coincides with the one which would be suggested by the phonological syllabification rules.

If only one hypothesis is accepted, the boundary can be predicted with certainty, if not, the acceptable hypothesis first pushed on the stack is the most probable, but its correctness cannot be guaranteed.

Both DVP and BHG use the phonotactic data discussed in section 4.1, extended with lists of possible syllable-final clusters of length one, two, three and four (the maximum), and with a list of possible syllable-initial clusters of length one. Unfortunately, anything goes in this case, so checking this list does not make very much sense. This additional information can be found in Table 1. As in section 4.1, the data are based on the statistical study by Brandt Corstius (1970), modified by our own data.

Head, middle with inserted boundaries, and tail are brought together again in the output. Two modes are possible: in *c-mode* (cautious mode), only boundaries with absolute certainty are kept in the output, in *d-mode* (dare-devil mode), the most probable alternative is presented if no absolute certainty can be attained. E.g.: *postzegel* (stamp) is hyphenated *post-zegel* in c-mode, and *post-ze-gel* in d-mode. The use of d-mode, of course, clashes with the philosophy behind CHYP, but we shall see later on that this mode can be profitably used in a number of applications.

Clusters of one non-syllabic segment:

Any non-syllabic segment except v, h, y, z, j, and qu.

Clusters of two non-syllabic segments:

lt, ld, ls, lp, lk, lf, lg, lm, mt, md, ms, mp, mb, mf, nt, nd, ns, xt, nx,
nk, sch, ng, nc, rc, rm, rn, rt, rd, rs, rp, rk, rf, rg, pt, ps, ts, kt,
ks, wt, wd, ws, ds, ft, fd, fs, gt, gd, gs, st, sd, cht, chs, ct, sp, sk.

Clusters of three non-syllabic segments:

lms, lmd, lst, lfs, lft, lpt, lps, lgt, lgd, lks, lkt, ldt,
lds, rmd, rms, rns, rts, rds, rdt, rst, rft, rfd, rps, rpt,
rgs, rgd, rgt, rks, rkt, rcht, mbt, mpt, mst, ndt, nds, nts,
nct, nst, nkt, nks, ngt, ngd, ngs, gst, kst, tst, fst, chts, chst.

Clusters of four non-syllabic segments:

mbts, ngst, ndst, rnst, chtst, rfst, lfts, ktst.

Table 1: Restrictions on the form and length of the coda in Dutch;
possible syllable-final consonant clusters

Statistical Data. In order to be practical, the probability that CHYP refuses to split up a word must be sufficiently low. We tested the program on all polysyllabic words starting with *a, w and z* present in the 'Uit den Boogaart corpus' (1975). The choice of the starting letters was random. The results are summarised in Tables 2 and 3.

	Number of Words	Number of Boundaries	Predicted Boundaries	Wrong boundary due to morphology (in d-mode)
A	2653	6904	2298 (33.3%)	346 (5.0%)
W	1729	3552	1235 (34.8%)	218 (6.1%)
Z	1164	2191	742 (33.9%)	143 (6.5%)
Total:	5546	12647	4275 (33.8%)	707 (5.6%)

Table 2: Results of applying CHYP to a corpus of polysyllabic words.
The letters A, W and Z of the Uit Den Boogaart corpus were used as data.
Number of boundaries, number of these found with absolute certainty by CHYP,
and number of errors made by CHYP in d-mode are given.

According to these data, some 34% of all syllable boundaries can be predicted by means of the phonotactic restrictions mentioned, and in almost 15% of polysyllabic words, *all* hyphens could be predicted.

Number of Syllables	Number of Words	%	Number of words Completely Predicted	%	At least one Boundary Predicted	%	At least two Boundaries Predicted	%
1	340	6.13	(340	100.00)				
2	1212	21.85	513	42.33	513	42.33		
3	1752	31.59	215	12.27	1076	61.42	215	12.27
4	1384	24.95	33	2.38	971	70.16	324	23.41
5	585	10.55	3	0.51	456	77.95	203	34.70
6	210	3.79	1	0.48	180	85.71	103	49.05
7	54	0.97	0	0.00	51	94.44	28	51.85
8	7	0.13	0	0.00	7	100.00	5	71.43
9	2	0.04	0	0.00	2	100.00	2	100.00
Totals:	5546		765 (14.7%)		3256 (62.5%)		880 (22%)	

Table 3: Input words separated according to number of syllables. Number of words for each class of syllable-length, number of words of which all syllable boundaries were predicted, number of words of which at least one syllable boundary was predicted, and number of words of which at least two syllable boundaries were predicted.

Of all polysyllabic words analysed, 63% obtained at least one boundary on phonotactic grounds. This may seem a prohibitively low percentage, but 72% of words with a length of more than two syllables obtained at least one phonotactically motivated hyphen, and 82% of words with a length of more than three syllables.

It seems reasonable to expect that these results can be improved by looking for further phonotactic restrictions. Besides restrictions on the length and form of syllable-initial (onset) and syllable-final (coda) clusters, there are also restrictions on the combination of nucleus and coda of a syllable and on the form of the nucleus. E.g., the coda *ng* cannot occur in combination with a nucleus consisting of a long vowel (**eeng*, **oong* etc.). And a nucleus consisting of long vowel <uu> or <aa> cannot co-occur with an empty coda (cp. section 4.1.1). Using this information, the syllable boundary in a word like *maatregel* (measure) can be predicted with certainty, because *maa-tregel* is excluded. Restrictions like these, which can be found with the help of statistical studies of the spelling syllable and phonological syllable, could be added to the hyphenation system.

One important problem which confronts anyone trying to collect a set of restrictions is the *dilemma between completeness and efficiency*. In order to have a system which makes no mistakes, it is necessary to include only those restrictions which are absolutely exceptionless. On one occasion, we departed from this self-imposed guideline, by removing *z* from the list of possible codas. In practice, this means that

whenever a *z* is encountered in an intervocalic cluster, a hyphen can be inserted before it. As far as we are aware, there is only one (loan) word in Dutch which violates this constraint: *fez*. This implies that possible compounds with *fez* might induce the program to put a hyphen in the wrong place. On the other hand, if the restriction were removed, the overall performance of the program would decrease considerably. It seems difficult to determine how far one can go in sacrificing theoretical purity under the pressure of efficiency. It would be foolish to lose a hundred in order to save one, but this utilitarianism opens the door to ad hoc rules and solutions.

Evaluation. A system like CHYP, which purports to be near to infallibility, makes sense only when it almost never makes a mistake. Of the 4,275 syllable boundaries predicted by the system in c-mode, 15 were wrong (0.35%). Four errors were not due to the system, but to incomplete input (the omission of a hyphen or a diaeresis). We list them with their faulty syllabification: *zeventiendee-euwse* (seventeenth-century), *zielee-enheid* (unity of soul), *zoe-ven* (a moment ago) and *aw-wuitkering* (unemployment benefit; AWW is an abbreviation). Subtracting these cases brings the error rate down to 0.28%.

The system hyphenated two other abbreviations: *a-ow* and *ak-zo*. Abbreviations should be entered in an exception list. Another possibility is to add a global condition to the program preventing hyphenation of words less than five characters long to avoid this problem. The hyphenation of short words is never very aesthetic, anyway (see stylistic rule number 6 in section 5.2.2).

The remaining nine errors are all foreign words (in this case English ones): *abortuste-ams* (abortion teams), *Avenu-es-hop* (Avenue shop), *warhe-ads*, *Washington*, *whiteco-ats*, *wors-hip*, *wervingste-am* (recruitment team) and *weduwes-hagjes* (little widow-shag). Four of these errors are due to the omission of *sh* as a possible syllable-initial cluster in our phonotactic data. If we add this cluster, the number of unavoidable errors totals five (0.12%). We believe that a success rate of 99.88% can be called acceptable.

The only way to prevent the remaining mistakes is to store all frequently used foreign words in a lexicon with their proper hyphenation, and check this lexicon before applying the hyphenation rules.

5.2.3.2 Optimising the Interaction with Analysis

Measured in processing time, morphological analysis is an expensive operation. Furthermore, there is a small possibility of errors due to the faulty operation of the word form parser or the affix analysis. The source of these errors is mostly the incompleteness of the lexical database used by the parser. Therefore, it would be useful to avoid morphological analysis in those cases where it is not absolutely necessary. This would minimise the chance of transporting errors from the morphological part into the syllabification part of the algorithm, and reduce the total cost in processing time.

A solution would be to build a program which first tries to find a hyphenation on the basis of phonotactic restrictions (using CHYP), and which transfers control to the main syllabification program (including morphological analysis) whenever it fails.

Unfortunately, morphological analysis can only be avoided if all syllable boundaries of a word can be predicted. Whenever there is a boundary which cannot be predicted, the whole word must be morphologically analysed. An example is *ont#wikkel* (to develop): the first boundary cannot be predicted, only the second one (*ontwik-kel*). The whole word must be morphologically analysed to know the proper hyphenation. Nevertheless, on the basis of the statistical test described earlier, it was calculated that in some 15% of poly-syllabic words, all syllable boundaries can be predicted by means of a simple set of phonotactic restrictions, making morphological analysis superfluous. This implies a considerable gain in processing time which more than counter-balances the overhead of using CHYP in vain in 85% of the input. However, a variant of CHYP should be used in that case which gives up when the first non-predictable syllable boundary is encountered, instead of trying them all.

In a superficial sense, this approach resembles a second generation expert system (Steels, 1985; Steels and Van de Velde, 1985). Phonotactic restrictions can be interpreted as heuristics (*shallow knowledge*) which can give a solution to a problem quickly, but are restricted in application. Syllabification rules using morphological analysis, dictionary lookup and phonological knowledge, constitute *deep knowledge* on which a program can fall back whenever the heuristics fail. The main difference with second generation expert system technology is that in this case heuristics cannot be derived automatically from the deep knowledge because the deep and the shallow knowledge are of a qualitatively different nature. We will have more to say about second generation expert systems in the context of Intelligent Computer Assisted

Instruction (chapter 7).

5.2.3.3 CHYP as an Autonomous System

Two variants in which CHYP could be applied as an autonomous system come to mind: a fully-automatic aid in text-formatting, and a semi-automatic aid in interactive hyphenation.

Partial Automatic Hyphenation in Justification. In applications like automatic hyphenation during justification, where not every syllable boundary needs to be computed — a maximum of one boundary per word is needed, and not every word must be hyphenated —, CHYP could be used to exclude morphological analysis altogether, and with it the need for a large lexical database in external memory. The data in Table 3 seem to offer hopeful prospects that CHYP can indeed predict enough boundaries to have a useful effect on justification. Long words are the most likely candidates for hyphenation in a text formatting environment, and of more than 70% of words with more than two syllables, at least one boundary can be predicted.

To produce an optimal effect, an intelligent interaction between the partial hyphenation algorithm and a justification algorithm seems necessary. We have not worked out such an interaction yet, but to test its usefulness we used the CHYP program in combination with the RUNOFF formatting system of Digital Equipment Corporation. RUNOFF incorporates no hyphenation algorithm, but allows the user to specify places in a word where it can be split (soft hyphens). The justification program uses this information. In the test, the syllable boundaries were computed by CHYP. Figure 3 lists a randomly chosen paragraph of Dutch text, the result of applying CHYP to it, and two formatted versions; the second one uses partial hyphenation.

The success of a justification algorithm can be measured by counting the number of spaces in the text. Ideally, the number of spaces per line should be equal to the number of words (or word parts, a hyphenated word consists of two word parts, separated over two lines) on that line minus one (disregarding typographical nuances like two spaces after a full stop). Consequently, a 'spacing-index' (SI) like the one given in formula (1), gives an estimate of the success of the justification of a paragraph.

```
(i) Input text

Zelfs goedkope microcomputers kunnen tegenwoordig
uitgerust worden met programmatuur die de gebruiker
in staat stelt de spelling van teksten automatisch te
corrigeren. In dit verslag willen we enkele gebruikelijke
algoritmen bespreken en evalueren, en ook nagaan hoe
taaltechnologisch onderzoek kan bijdragen tot het
ontwikkelen en
perfectioneren van dergelijke algoritmen.

(ii) Output of CHYP (16 boundaries predicted, 55.6% of polysyllabic words
     received at least one syllable boundary)

Zelfs goed=kope microcomputers kun=nen tegen=woordig
uit=gerust worden met program=matuur die de gebruiker
in staat stelt de spel=ling van teksten automatisch te
cor=rigeren. In dit verslag wil=len we enkele gebruikelijke
algorit=men bespreken en e=valu=eren, en ook nagaan hoe
taaltech=nologisch onder=zoek kan bijdragen tot het
ontwik=kelen en
perfecti=oneren van dergelijke algorit=men.

(iii) Formatted text, CHYP-output not used (spacing-index = 2.07)

Zelfs   goedkope   microcomputers   kunnen   tegenwoordig
uitgerust  worden  met  programmatuur die de gebruiker in
staat  stelt  de  spelling  van  teksten  automatisch  te
corrigeren.     In    dit    verslag    willen    we    enkele
gebruikelijke algoritmen bespreken en evalueren,  en  ook
nagaan  hoe taaltechnologisch onderzoek kan bijdragen tot
het  ontwikkelen   en   perfectioneren   van   dergelijke
algoritmen.

(iv) Formatted text using CHYP-output (spacing-index = 1.27)

Zelfs goedkope microcomputers  kunnen  tegenwoordig  uit-
gerust worden met programmatuur die de gebruiker in staat
stelt de spelling van teksten automatisch te  corrigeren.
In  dit verslag willen we enkele gebruikelijke algoritmen
bespreken en  evalueren,  en  ook  nagaan  hoe  taaltech-
nologisch  onderzoek kan bijdragen tot het ontwikkelen en
perfectioneren van dergelijke algoritmen.
```

Figure 3: A Dutch paragraph, as entered in the editor (1), with syllable boundaries determined by CHYP indicated (2), in a justified version with Digital Runoff system (3), and in a justified version using Digital Runoff in combination with the output of CHYP.

(1) $$SI = B/(W - L) \quad (\text{with } W > L)$$

This formula expresses the ratio of the number of blanks (B) to the number of word parts (W) minus the number of lines (L) in a particular paragraph. The nearer SI approaches one, the more successful the text-formatting. When SI equals one, there is exactly one blank between every two consecutive word(part)s on a line in the paragraph. A more sophisticated measure of white space should be used to evaluate

justification in systems which work with proportional character widths (variable width fonts).

A randomly chosen Dutch text, with a length of 797 words (considered here one paragraph) was processed with and without CHYP for 25 different line-lengths between 20 characters and 120 characters. The results are in Table 4 and Figure 4.

Line-length	SI (no hyphenation)	SI (CHYP)	Gain	SI (complete)	Gain
120	1.23	1.22	0.8	1.12	9.0
115	1.34	1.27	5.2	1.12	16.4
110	1.26	1.21	4.0	1.12	11.1
105	1.41	1.21	14.2	1.11	21.3
100	1.38	1.29	6.5	1.14	17.4
95	1.43	1.32	7.7	1.15	19.6
90	1.42	1.23	13.4	1.15	19.0
85	1.45	1.34	7.6	1.17	19.3
80	1.53	1.26	17.7	1.16	24.2
75	1.51	1.38	8.6	1.19	21.2
70	1.55	1.30	16.1	1.24	20.0
65	1.65	1.49	9.7	1.22	26.1
60	1.67	1.49	10.8	1.25	25.2
55	1.59	1.53	3.8	1.31	17.6
50	2.01	1.54	23.4	1.34	33.3
45	1.94	1.69	12.9	1.35	30.4
40	2.08	1.72	17.3	1.43	31.3
35	2.28	1.91	16.2	1.47	35.5
33	2.42	1.88	22.3	1.56	35.5
31	2.59	1.96	24.3	1.49	42.5
30	2.59	2.13	17.8	1.57	39.4
29	2.75	2.14	22.2	1.61	41.5
27	2.95	2.23	24.4	1.67	43.4
25	3.07	2.43	20.9	1.69	45.0
20	4.02	3.03	24.6	2.01	50.0
		Average Gain:	14.1	Average Gain:	27.8

Table 4. Spacing index SI for a random text with different line-lengths, without hyphenation, with CHYP, and with complete hyphenation, respectively. The percentages represent the gain (reduction of SI) as compared to filling without hyphenation.

The average gain (measured in the reduction of SI) was 14.1%. The gain with the complete hyphenation system incorporating morphological analysis, described in section 5.2.2, averaged 27.8% (this constitutes the upper bound). The minimum gain with CHYP was 0.8% (for line-length 120) and the maximum gain 24.6% (for line-length 20). The fact that these values are the extremes of the sequence of line-lengths tested seems to be coincidental since high and low values for SI occur with other line-lengths as well. This high standard deviation shows that the success of CHYP is determined partly by the words which happen to be a candidate for hyphenation. However, there is a clear non-coincidental tendency for the reduction of SI to be

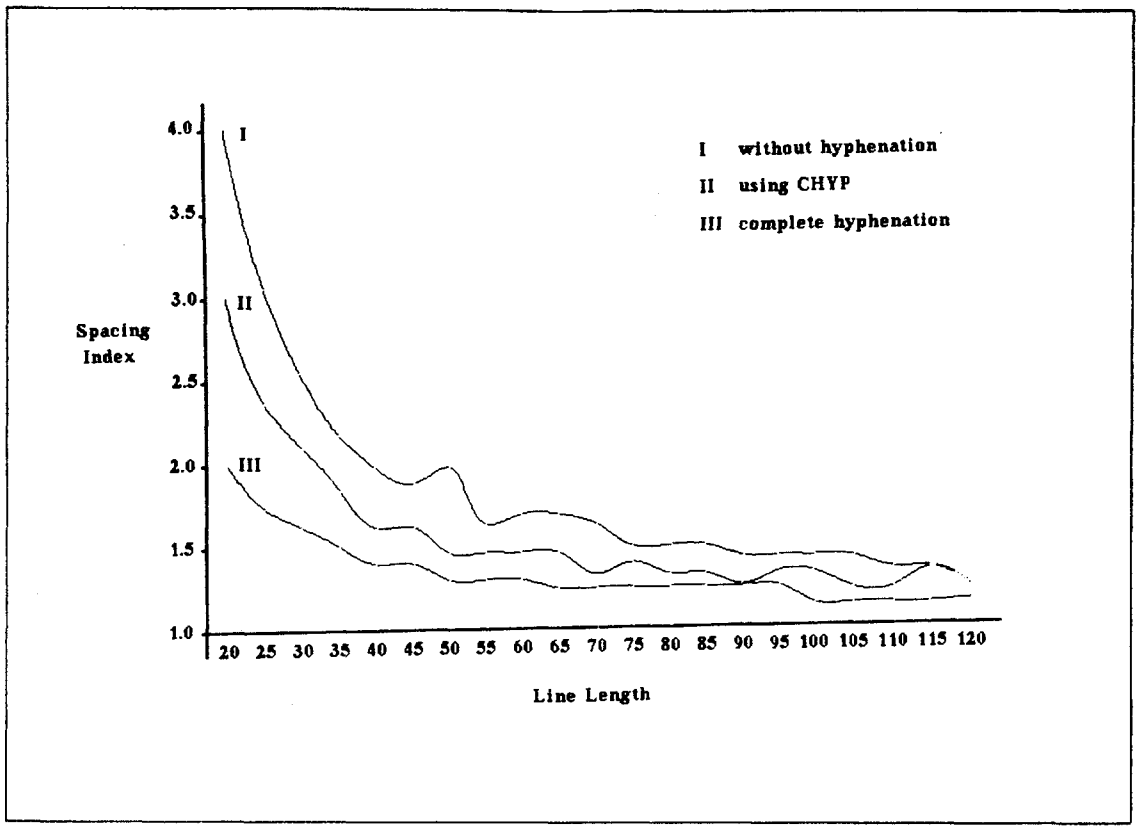


Figure 4. Based on Table 4: path of SI without hyphenation, SI using CHYP and SI using complete hyphenation as a function of line-length.

highest with small line-lengths (between 20 and 35). This is fortunate, as most newspapers and magazines use small column widths. Next page shows some sample output of the filling algorithm: without hyphenation, using CHYP and using full hyphenation, respectively.

(c-mode)
NIL
(write)
Er zijn twee soorten woordenboeken: STAMMEN-
WOORDENBOEKEN en vormenwoordenboeken. In het
eerste geval nemen we alleen de stam van een
woord op als trefwoord. Bij een vormenwoordenboek
representeren we alle verbogen, vervoegde en
samengestelde vormen van een woord als een apart
trefwoord. Als we een stammenlexicon gebruiken,
moet het programma alle afgeleide vormen zelf KUN-
NEN berekenen, anders zou de detector vele COR-
RECTE woorden aanduiden als fout. We hebben GEKO-
ZEN voor een volledig vormenwoordenboek. Een voor
de computer leesbare versie van de nieuwe Van
Dale voor het hedendaagse Nederlands was ons UIT-
GANGSPUNT. Op dat woordenboek hebben we een
programma losgelaten dat automatisch alle AF-
GELEIDE vormen berekent. Het resultaat was een OP-
GEBLAZEN Van Dale met meer dan een half miljoen
woordvormen. Voor een taal als het Engels zou een
vocabularium met die omvang al meer dan voldoende
zijn om een efficiente detector te bouwen.
Spacing index:
1.453125

(d-mode)
NIL
(write)
Er zijn twee soorten woordenboeken: STAMMENWOOR-
DENBOEKEN en vormenwoordenboeken. In het eerste
geval nemen we alleen de stam van een woord op
als trefwoord. Bij een vormenwoordenboek REPRESEN-
TEREN we alle verbogen, vervoegde en SAMENGESTEL-
DE vormen van een woord als een apart trefwoord.
Als we een stammenlexicon gebruiken, moet het PRO-
GRAMMA alle afgeleide vormen zelf kunnen BEREKE-
NEN, anders zou de detector vele correcte woorden
aanduiden als fout. We hebben gekozen voor een
volledig vormenwoordenboek. Een voor de computer
leesbare versie van de nieuwe Van Dale voor het
hedendaagse Nederlands was ons uitgangspunt. Op
dat woordenboek hebben we een programma LOSGELA-
TEN dat automatisch alle afgeleide vormen BERE-
KENT. Het resultaat was een opgeblazen Van Dale
met meer dan een half miljoen woordvormen. Voor
een taal als het Engels zou een vocabularium met
die omvang al meer dan voldoende zijn om een EFFI-
CIENTE detector te bouwen.
Spacing index:
1.3100775

Lisp Listener 2

(hyphenation nil)
NIL
(write)
Er zijn twee soorten woordenboeken:
stammenwoordenboeken en vormenwoordenboeken. In
het eerste geval nemen we alleen de stam van een
woord op als trefwoord. Bij een vormenwoordenboek
representeren we alle verbogen, vervoegde en
samengestelde vormen van een woord als een apart
trefwoord. Als we een stammenlexicon gebruiken,
moet het programma alle afgeleide vormen zelf
kunnen berekenen, anders zou de detector vele
correcte woorden aanduiden als fout. We hebben
gekozen voor een volledig vormenwoordenboek. Een
voor de computer leesbare versie van de nieuwe
Van Dale voor het hedendaagse Nederlands was ons
uitgangspunt. Op dat woordenboek hebben we een
programma losgelaten dat automatisch alle
afgeleide vormen berekent. Het resultaat was een
opgeblazen Van Dale met meer dan een half miljoen
woordvormen. Voor een taal als het Engels zou een
vocabularium met die omvang al meer dan voldoende
zijn om een efficiente detector te bouwen.
Spacing index:
1.5950413

An Interactive Hyphenation Program. The second application of CHYP which suggests itself is an interactive hyphenation program. In this case CHYP is forced to suggest a syllable boundary even when it is not absolutely sure (d-mode is used). Recall that in the latter case, the most likely position (as regards the general syllabification rules discussed in section 4.1) is presented. A 'normal' performance rate of about 95% can be expected this way, with errors due to the interference of morphological structure (especially word boundaries in compounds and after prefixes). However, the strength of CHYP is that it also keeps the alternative possibilities (never exceeding three) available when it suggests an 'uncertain' hyphen. A human operator noticing an error, simply touches one key or clicks once with his mouse, and CHYP presents its second best guess. Considering the facts (a) that the user is only confronted with a hyphenation when the justification algorithm wants a word split, (b) that only in some five percent of these cases an error results, (c) that correction of these errors involves only minimal effort, and (d) that the system is extremely compact in comparison to most existing systems[4], CHYP seems a good solution to the hyphenation problem when user feedback is available. Even more so when a system with morphological analysis is not possible due to storage problems or for reasons of efficiency.

When using CHYP as an autonomous system, its performance can be improved further by incorporating probabilistic data. The various hypotheses accepted by the Boundary Hypothesis Generator could be provided with a probability measure based on statistical data. These measures could be modified further using the feedback provided by the user (whenever a suggestion by CHYP turns out to be wrong, its probability drops). This facility is not yet implemented. Another modification which we did implement is to comform the behaviour of CHYP in d-mode completely to the phonological rules of section 4.1.2 (i.e. also in the case of clusters of length two). This way, the success-rate of the program in d-mode can be raised to about 97%, but not very much higher; this coincides with the success-rate attained by current hyphenation programs (often using large amounts of patterns or large exception dictionaries).

[4] A prototype written in Franz Lisp occupies 12K internal memory on a SUN Workstation, and splits words in real time. Most existing systems contain large exception dictionaries which occupy a lot of memory.

5.2.4 Other Approaches to Dutch Hyphenation

In what follows two traditional approaches to Dutch hyphenation, one based on list processing (Brandt Corstius), the other on pattern matching (Boot), will be discussed. We will show that Boot's approach, though it may have practical results, is not valid in principle. Brandt Corstius' approach on the other hand, is valid in principle, but invalid in elaboration. Both approaches are inferior to the ones presented earlier in this chapter (sections 5.2.2 and 5.2.3).

5.2.4.1 Brandt Corstius

The approach taken by Brandt Corstius (1970) in his SYLSPLIT system is comparable to the hyphenation strategy in section 5.2.2 because it includes a primitive[5] form of affix-stripping (without dictionary). In SYLSPLIT, however, morphological analysis as a concept is absent, and is in practice limited to prefixes and suffixes. Furthermore, affixes are taken in a broad, non-linguistic sense; lexical morphemes like *hoofd* (head), *land* (country), and *groot* (great) are analysed as affixes. Suffix analysis precedes the syllabification rules, prefix analysis is intertwined with them. Due to this mixing of levels, a number of ad hoc rules is necessary to boost the performance of the program. E.g., since *der* is a prefix, and prefix-analysis is intertwined with the syllabification rules, *studeren* (to study) is split incorrectly as *stu-der-en*. To prevent this, *ren* was entered in the list of suffixes although it is clearly not a suffix.

The fact that there is no analysis of compounds generates other ad hoc rules and exceptions. E.g.: *'When s follows ng, it belongs to the same syllable'*. This strange rule was entered to prevent the wrong syllable split of words like *regering-sonderhandelingen* (government negotiations). It generates new errors in words like *lang-ste* (longest).

It is clear that in practical use, a growing number of ad hoc rules, exceptions and nonce-affixes will make the program opaque and its results unpredictable. A program based on this approach will never be able to hyphenate unrestricted text with a performance of more than about 97% (data by Brandt Corstius), and will need continuous (manual) updating of exception lists. It is basically a list-processing ap-

[5] Proper affix-analysis is impossible without a dictionary (cf. 3.2.).

proach, and all that can be done is adding elements to the lists and changing the order in which the program makes use of them. Fundamental improvements are impossible within such a framework.

Our own program, although following roughly the same philosophy, is much more surveyable. By separating morphological analysis from syllabification, a modular architecture is obtained. By following an 'only-if-needed' strategy in the interaction between morphological analysis and syllabification, the intrusion of errors in the syllabification algorithm due to morphological analysis is minimised, and the output of the program is largely predictable. No ad hoc rules are necessary. Moreover, the distinction between cohesive and possible clusters, absent as a concept in SYL-SPLIT, proves to be effective.

5.2.4.2 Boot

The approach taken by Brandt Corstius has been termed 'first-generation' by Martin Boot (1984), and has been severely criticised by him (he describes it as a 'bite off and gamble' algorithm). However, most objections he raises against this approach (unpredictable results, ad hoc rules and affix lists) are no longer applicable to our variant. Indeed, we believe that our system compares favourably to the alternative HUPHE system he describes as being 'second generation'.

Boot argues that people do not consult an internal dictionary to divide words into syllables. They would have certain expectancies instead, on the basis of which letter and sound patterns are recognised. For instance, in splitting up *ontheologisch* (untheological), people do not recognise the existence in their mental dictionary of the prefix *on-* (meaning not) and the word form *theologisch*, they rather know that no word in Dutch can start with the pattern *heol*. Accordingly, the HUPHE system uses no dictionary and consequently no morphological analysis is present.

> 'Morphology as a descriptive system offers no reliable basis for computer programs for hyphenation (Boot, 1984:24) [my translation, WL].'

We hope to have made clear in section 4.1. that this statement is plainly false; morphology as a descriptive system makes a crucial difference between morpheme and word boundaries. Boot only considers the 'innocent' type (the morpheme boundary, as in *lop+en*, to run), which never overrules syllabification rules, and is therefore indeed unimportant for syllabification.

HUPHE consists of a pattern recognition system, and a set of some 1300 patterns, from which syllable boundaries can be derived. In a sense, each pattern defines a syllable boundary. E.g. the pattern (* (4TSN) (123 - 4) *) defines the syllable boundary in words containing ...ietsn..., such as *nietsnut* (good-for-nothing). The character '4' stands for the spelling string *ie*. The string *123-4* signifies that a hyphen is to be inserted between the third and the fourth character of the pattern. Maximum length of a pattern is six letters[6] (the difference with a dictionary entry becomes vague here). The order in which the system searches for these patterns is sometimes important, too (namely for the vowel patterns). The pattern inventory was obtained by trial and error and seems to need continuous updating and modification, on the basis of test results. This is hardly surprising. Apart from the phonotactic constraints on onset and coda discussed earlier, and other, phonologically motivated restrictions, any syllable boundary predicting pattern is ad hoc and necessarily provisional.

This is due to the fact that in Dutch the number of potential words is infinite in principle. The parts of a compound are concatenated in spelling. Compare Dutch *computerprogramma* to English *computer program*. In the latter language, the parts of a compound are separated by a space or by a hyphen. It follows that in Dutch any combination of a possible syllable-final cluster and a possible syllable-initial cluster is allowed, and may at any time be realised. An example (borrowed from Boot) will clarify this point. Take a word like *klimop* (ivy). In our system, this word is analysed by the parser (*klim#op*), the word boundary is replaced by a syllable boundary, and a correct hyphenation is returned. In the HUPHE system, this word was split up wrongly (*kli-mop*), and consequently, a new minimal pattern was looked for. First *-op* was considered, but this would cause another error in words like *katjesdrop* (licorice). The second hypothetical minimal pattern was *m-op* generating errors in compounds with *mop* (joke). Next pattern was *im-o*, but this was impossible because of *lim-onade* (lemonade). Finally, Boot decided to use *im-op* as a minimal pattern. However, we had no difficulty in immediately finding a compound that would be hyphenated wrongly by this pattern: *Eskim-opaar* (Eskimo couple). A better minimal pattern might be *klim-o*, but again we found a counter-example in *aanmaaklim-onade* (syrup). After discarding *lim-op* as a pattern because of *li-mop* (joke about the Chinese linear measure *li*), only one solution was left: store the whole word as a

[6] It is not clear whether the term 'letters' refers to spelling symbols or to elements of a phoneme-like alphabet (e.g. 4 = ie), in which case patterns may actually be much longer than six spelling characters.

minimal pattern! This is clearly not what Boot wanted.

Only the phonotactic constraints described earlier can be trusted to be applicable to the whole of the Dutch vocabulary, because they are phonologically motivated. Any other restriction (be it in the form of a minimal pattern or in any other form) is ad hoc and has only limited usefulness.

Systems like HUPHE can be efficient as hyphenation programs for a subset of Dutch, but not for the whole potential vocabulary; a simple theoretical investigation of intervocalic consonant clusters makes this clear. Some $2,6 * 10^9$ different clusters are *theoretically* possible between two vocalic segments ($22^1 + 22^2 + ... + 22^7$). The general formula to calculate this number is $\Sigma_{i=1}^{n} K^i$, where K is the number of consonant phonemes and n the maximum number of consonant phonemes between two vowels (or, maximal onset length plus maximal coda length) in a language.

Using an inventory of possible syllable coda and onset structures, the number of *phonologically* possible sequences of consonants which can occur between two syllabic segments can be calculated (the number is 7841, or 0.0003% of what was theoretically possible)[7].

Not all these clusters are realised in any given corpus, but they are possible in principle, and may at any time be realised, notably in new compounds. In almost 90% of these clusters, the syllable boundary can be predicted without morphological analysis. Unfortunately (but logically), the number of predictable boundaries is smallest with those clusters occurring most frequently, i.e. those consisting of one or two non-syllabic segments. The shorter the cluster, the less predictable the syllable boundary. This contradicts Boot's (1984:20) statement:

> 'the more consonants between vowels, the higher the probability of errors in hyphenation' [my translation, WD]).

Quite the contrary is true: the longer the intervocalic consonant cluster, the easier it is to find a unique syllable boundary.

Table 5 and Figure 5 give some information about possible inter-vowel consonant clusters. Distribution and predictability percentage for clusters from length one

[7] This number was calculated by combining all possible onsets with all possible codas (based on our statistical study of the spelling syllable in section 5.2.3.1 and Chapter 4).

to seven are given in Table 5. Figure 5 pictures the relation between predictability and frequency. The data for frequency were obtained by analysing the intervocalic consonant clusters of 17,907 word forms from the *top ten thousand* dictionary of frequent Dutch word forms. The number of cluster types was 486 (6.2% of what is phonologically possible), and the number of cluster tokens 35,414.

Cluster Length	Number	%	Number with a Predictable Boundary	%
1	22	0.28	5	22.72
2	355	4.53	271	76.34
3	1529	19.50	1255	82.08
4	2860	36.47	2481	86.75
5	2355	30.03	2191	93.08
6	664	8.47	656	98.80
7	56	0.71	56	100.00

Table 5. For each cluster length, number of phonologically possible clusters is given, and the number and percentage of those the boundary of which can be unambiguously predicted.

For other languages, with a different compounding mechanism (like English), the situation may be different. The pattern matching approach of Boot is probably based on the work of Frank M. Liang at the university of Stanford in the late seventies (quoted by Knuth, 1979). Liang developed a program which computes a table on the basis of a dictionary of hyphenated words. Since the hyphens can be reconstructed from the table by means of a pattern matching algorithm, it suffices to store the table. The main difference with Boot's approach seems to be that in the latter patterms have to be introduced by hand.

The patterns thus collected seem to work for words which are not present in the dictionary as well, although an exception dictionary is needed. The system was incorporated in the T_EX type-setting program (Knuth, 1979). Knuth claims that the same program can be adapted for other languages by computing new patterns. We have shown that this is not true, because complete dictionaries of Dutch are impossible to construct (this argument applies to all languages which spell compounds as single words, e.g. German, Swedish, Norwegian, Danish and many others, but not to English and French).

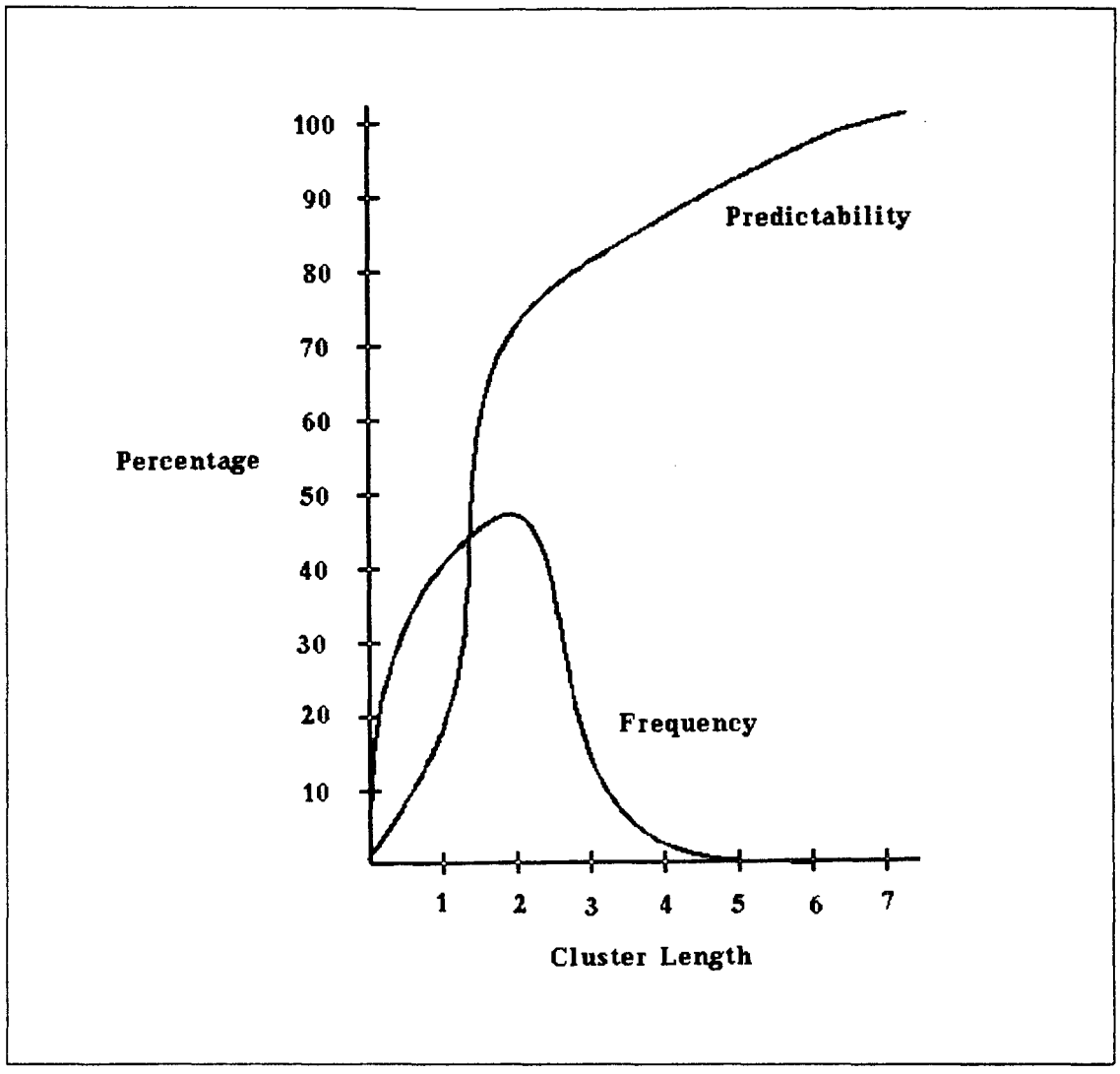


Figure 5. Predictability of phonologically possible intervocalic clusters in Dutch, based on Table 5, and measured frequency.

5.2.5 Some Residual Problems

(i) There is an element of unpredictability in the morphological analysis preceding syllabification. This was fully discussed in section 3.2. When morphological analysis is not used (as in CHYP), this source of errors disappears. When it is used only if necessary (as in the program incorporating morphological analysis), its disturbing effect is minimal.

(ii) Ambiguity cannot be resolved without semantic information. E.g.: *kwart-slagen* (quarter turns) versus *kwarts-lagen* (quartz layers), *ui-tje* (little onion) versus *uit-je* (outing), and *alm-achtig* (like a mountain meadow) versus *al-machtig* (all-mighty). This ambiguity is detected during morphological analysis. Two or

more parallel representations are kept by the system, but at the point when syllabification rules must apply, one possibility must be chosen. In the present system, this choice is made randomly. When integrated in an interactive text editor, e.g. an Author Environment, the program could ask the user for advice. Another possibility in some applications is to refuse hyphenation in cases of ambiguity. In any case, the number of instances of this kind of ambiguity is not very disturbing, examples are difficult to find.

(iii) When abbreviations contain vowel clusters without full stops, they are hyphenated according to the default rules. E.g., *aow* is hyphenated *a-ow*.

(iv) The most vexing problem left is the hyphenation of foreign words, which are numerous in Dutch scientific and newspaper language. They could be stored in the lexicon with a complete syllabification, or a hyphenation program for the language in question could be activated. If no lexicon is used, there is no way they can be detected, and the regular Dutch morphological and syllabification rules apply.

5.3 Conclusion

Author Environments promise to be tools of considerable usefulness to both writers and editors. A morpho-phonological module such as the one described in Part II will be an indispensable component of the linguistic knowledge and procedures incorporated in the AE. As regards automatic hyphenation, three main points were made in this chapter:

(i) Partial morphological analysis is necessary for correct syllabification of the complete potential vocabulary of Dutch (all existing and possible simplex and complex words). More precisely, those morphological boundaries which overrule syllabification rules must be recognised by morphological analysis and replaced by syllable boundaries, before these syllabification rules apply. This is what the system described in section 4.1, and taken up again in section 5.2.2, is capable of doing. If morphological analysis is not included, a gambling element is introduced into the algorithm (with a five percent chance of losing for each computed syllable boundary).

(ii) Phonotactic restrictions can make morphological analysis superfluous in some cases. On this phenomenon CHYP is based, an algorithm which only hyphenates when it is absolutely sure (exception made for foreign words, ambiguous words and abbreviations). In combination with a justification algorithm this extremely

simple algorithm improves the formatting capacities of text editors considerably.

(iii) The methodological point, implicit in this approach to hyphenation, is that linguistic knowledge (in the form of a lexicon and a morphological parser) is necessary to attain reasonable results, but that phonotactic information, collected through statistical analysis, may help to make an implementation more efficient, or may even suffice for restricted applications like the improvement of formatting systems, or hyphenation of restricted corpora. Furthermore, algorithms which are reasonable for one language, are not always reasonable for another. The pattern matching approach to hyphenation, for instance, is possible for English but not for Dutch.

CHAPTER 6

Automatic Detection and Correction of Errors

6.1 Background[8]

A poet can survive everything but a misprint.
Oscar Wilde

An important function of the Author Environment (section 5.1) is to detect and correct *errors*, which are defined here loosely as *unintentional deviations from some convention*. Different categories of errors can be distinguished; they are listed in Table 1. The Author Environment outlined in 5.1. can be equipped with programs which are able to detect and correct some of these errors automatically. These programs will be based on the linguistic algorithms described in Chapter 3.

There are two basic strategies possible in the development of a correction algorithm:[9] a knowledge-based and a statistical one. A *statistical approach* uses various quantitative techniques to correct errors; these may include frequency tables, trigram

[8] This section is based on Daelemans, 1984; Daelemans, Bakker and Schotel, 1984 and Daelemans, 1985.

[9] From now on we use the term correction to refer to both detection (verification) and subsequent correction of errors. These terms should not be interpreted in their psychology-of-writing sense. E.g. Bartlett (1982) distinguishes between *detection* ('there is something wrong here') *identification* (x is wrong because of y) and *correction* ('x should be z'). In a computer program the concept of *identification* is meaningless (if it is detected we know why, otherwise it would not have been detected). Similarly, the traditional distinction between *correction* (changing an evident error) and *revision* (changing something that was not necessarily wrong in order to improve it) is vacuous in this context.

```
Non-human
   Data-transmission     (Modem, telephone-line noise)
   Data-input            (OCR, speech input)
Human
(Due to ignorance or carelessness)
   Orthographical
     spelling            (mostly systematic)
     typing              (mostly unsystematic)
     hyphenation         (solved by preventing the author to
                          do it himself, see 4.1.1)
     punctuation
     lack of consistency in choice of spelling system
                         (Dutch enjoys the luxury of several spelling
                          systems)
   Morphological
     word-formation      (e.g. transition grapheme in compounds,
                          misderivations)
   Syntactic
     sentence construction
                         (e.g. subject-verb agreement)
   Discourse
     paragraph and text construction,
     logical and semantic
     pragmatic           (e.g. mixing of polite and familiar)
   Stylistic
     word, sentence, paragraph length
     choice of sentence constructions
                         (e.g. passive versus active)
     vocabulary          (e.g. abstract versus concrete)
     repetition
```

Table 1. A Typology of Word-processing Errors (not intended to be exhaustive or definitive)

tables, exhaustive dictionary lookup, etc. Such programs do not have any real linguistic knowledge but, owing to various programming 'tricks' like hashing, dictionary compression, hardware filters etc., they are reasonably efficient in accuracy and speed. Their application, however, is largely restricted to the correction of typing errors. This is the strategy used in almost all commercially available spelling checkers. In these systems, most effort is put in achieving an optimal compromise between accuracy, size of lexicon, processing time and storage prerequisites.

In a *knowledge-based strategy* it would be claimed that a program can only be sure about a mistake and how it is to be corrected if it understands the text. Therefore, explicit linguistic knowledge, including semantic and pragmatic knowledge, should be integrated into it. Indeed, even the division of words into syllables and of compounds into words cannot be completely automated without semantic knowledge (E.g. compare *valk#uil* (a kind of owl) to *val#kuil* (pitfall). To detect ambiguity and reasoning errors, the computer should even contain enough extra-linguistic (world) knowledge to have at least a vague notion of the intentions and world view of the au-

thor, and of the topic of the text. In short, a correction program would have to be a kind of expert system, integrating linguistic, world and domain knowledge. Such programs have not been developed yet, nor is it likely that they will be in the near future.

Again, as with the hyphenation problem, our approach will be knowledge-based (linguistic) in principle, with statistical methods to boost performance. We will restrict our attention to *word-level correction* (spelling and typing errors). Perhaps the most important distinction to be made among word-level errors is that between *spelling errors* (deviations from orthography due to the temporal ignorance of the writer) and *typing errors* (due to erroneous keystrokes by the writer). Although they share some characteristics (both types can be detected by a single algorithm), they are different in important respects (they cannot always be corrected by the same type of algorithm). We will come back to this distinction later.

The basic model to correct word level errors is a *two-stage model.* It consists of a detection module which takes a text as input and indicates the possible errors. A correction module then accepts these flagged words as input, and suggests one or more corrections. We will deal with these steps in turn.

6.2 Detection

The traditional algorithm for spelling error detection at the word level is simple. Every string of letters, separated by delimiter symbols (, . ; : ? ! <space>) is interpreted by the program as a possible word form. If such a possible word form can be found in a dictionary (a list of words) it is correct, if not it is a misspelling. Two major shortcomings are inherent to this approach:

[i] The problem of *undershoot* (failure to detect errors). A spelling error present in the word list is not detected. This is the case whenever a formally acceptable word violates semantic or syntactic restrictions, i.e., when the context determines whether a possible word is correct or wrong. For example in the phrase *I was reading the nail* (a semantic error), *nail* is not interpreted as an error because it is a valid English word. Similarly, with the phrase *I goes* (a syntactic error), the system again fails to detect the mistake, because there is nothing formally wrong with *goes.* The problem of undershoot cannot be overcome by word-level algorithms. A simple surface syntactic parsing would suffice to cope with morphological and syntactic errors but the system would have to be imbued

with world knowledge to make the detection of semantic errors possible.

[ii] The problem of *overkill* (flagging correct words as errors). Nothing is more irritating than a program which makes a habit of rejecting correctly spelled words. This problem occurs whenever a correct word is not present in the word list. At first sight, it would seem that, given enough memory facilities, this problem can simply be solved by storing enough words and keeping track of new words. Actually, this is what is done in most English versions of spelling checkers. But as we noted in Chapter 5, languages like Dutch and German allow for a virtually unrestricted concatenation of words into compounds which are spelled as single words (without spaces or hyphen), e.g. *spelfoutendetectieprogramma* (spelling error detection program). Such tapeworms which are generated freely and in large quantity in any style of text, cannot be fully represented in the word list of the spelling checker, and the formation of new ones cannot be predicted. In English, the parts of a compound are usually separated by a space or a hyphen. In French, compounds are avoided by using prepositional noun phrases. E.g. compare English *spelling error* and French *faute d'ortographe* to Dutch *spelfout*.

This example makes clear that natural language software must be redesigned for each language independently, even at this level. Spelling checkers for English are not readily transportable to other languages. A detection algorithm for Dutch should therefore combine the basic dictionary lookup strategy with a means of reducing compounds to their parts. To this end, the program must be extended with linguistic knowledge, namely knowledge about the rules which describe how new words can be formed from existing words (morphological knowledge). In section 6.2.1, we will describe a program which incorporates this knowledge by means of the word form parser and the lexicon of Chapter 3.

There is a problematic relation between undershoot and overkill: the problem of overkill can be diminished by making the dictionary larger, but a larger dictionary necessarily increases the probability of undershoot. Peterson (1986) has calculated that the probability of undershoot in systems with long word lists may approach one typing error out of six. Small topic-specific word lists could minimalise the occurrence of undetected typing errors.

6.2.1 DSPELL: Verification with an Unlimited Vocabulary

DSPELL was written in Franz Lisp and runs on VAX11/780 and SUN workstations. It can be used with UNIX formatting systems such as Runoff, Nroff and Troff, and is integrated into a screen editor (EMACS).

The system can be regarded as the detection module in a two-stage correction system: it finds spelling errors in Dutch text, and outputs them to another module which suggests corrections. We implemented only the detection module. The detection algorithm consists of three parts: reading, detecting and displaying. In the reading phase, the input text is transformed into lists of 'possible word forms'. This involves using punctuation and other information to isolate word forms. Furthermore, special symbols (e.g., formatting codes) and numbers must be omitted as it makes no sense to check them for errors. In the detection phase, it is decided for each isolated possible word form whether it is correct or incorrect. This is done by looking it up in a system of dictionaries. If a word form is not found in these dictionaries, morphological analysis is attempted: the system tries to break down compounds into their parts. If this fails as well, the word form is considered to be an error. In the display phase, the results of the detection are presented to the user or to an artificial correction module. We will go into the different phases in somewhat more detail.

In the *reading* stage, the input file is read line by line. Each line is represented internally as a list of atoms.[10] The main problem in this conversion is the interpretation of special symbols. Either they are transformed into a space, e.g. *auto-ongeluk* (car accident) is interpreted as two words (auto ongeluk), or they disappear: *^&woord\&* (^, & and \ are text formatting codes of the Runoff formatting system) becomes (woord). The fact that the hyphen '-' is transformed into a space implies that the two parts of a hyphenated word are interpreted by the system as two separate words. This often leads to errors. The problem is due to the ambiguous meaning of the hyphen in Dutch spelling (it indicates both hyphenation and compounding). As the Author Environment is equipped with a program that hyphenates automatically (section 5.2), the problem should not be too acute. For representing soft hyphens,[11] another symbol than the hyphen should be chosen. The correct interpretation of formatting codes boils down to the incorporation of a formatting command parser in the

[10] Atoms and lists are the primitive data-types of the programming language Lisp.

[11] Hyphens provided by the user to help the formatting system.

part of the program doing the reading. E.g. in Troff-like formatters, lines beginning with a dot are simply omitted, and the program knows which escape sequences (commands) are possible within words in order to remove them.

The *detection* phase combines dictionary lookup with morphological analysis as described in section 3.2. A short overview will refresh the reader's memory:

> Morphological Analysis consists of:
>
> i. Segmentation (find all possible morpheme distributions), which implies a number of dictionary lookups.
>
> ii. Parsing (keep those distributions that are permitted by the morphological grammar or signal an error if no analysis is found).

Morphological analysis makes the vocabulary of the detection system virtually unlimited and solves the problem of 'compound overkill'.

It is impossible to store all scientific terms of all disciplines, and all proper names in the dictionary. Besides, this would unnecessarily increase the probability of undershoot, as was pointed out earlier. To overcome the problem of the system flagging these words as errors, an initially empty dictionary is available where the user can easily enter these and other words not present in the main dictionary. The efficiency of the whole detection procedure is determined largely by the presence and completeness of this lexicon, especially for detecting errors in scientific text. The user lexicon of DSPELL contains most commonly used abbreviations and a virtually complete list of linguistic terminology. It can be extended unrestrictedly. It suffices to enter the citation form and exceptional derived forms. The system then computes the other (regular) derived forms, asks the user whether they are correct, and if so adds them to the user dictionary (see section 3.1). Forms can be removed from this dictionary as well. The user defined lexicon is checked before the main dictionary. It is even possible in principle to have a 'library' of user dictionaries, one for each discipline or domain of discourse. The user could then make a selection of these before starting the detection procedure. (Cf. Peterson, 1980a and 1980b).

In most texts, every word token has an almost 50 percent chance of being repeated at some place later in the text. Many detection algorithms therefore store words already processed in a separate dictionary which is checked before the others. In DSPELL, the same effect is obtained with Lisp property lists: each analysed word type gets a property 'already analysed' the value of which is + (correct the last time), - (flagged as an error last time), or NIL (not yet analysed). This property is

checked before any lookup operation is performed, and if the word was already analysed, the same result is returned as was computed the first time. Of course, this is possible only in a context-free algorithm. We did not provide a separate dictionary with the 500 most frequent words[12] of Dutch for storage in internal memory (to improve efficiency). The main lexical database used, though in external memory, can be accessed fast enough for on-line verification.

During the *display stage*, in the absence of a correction procedure, the system returns a list of hypothetical errors. The flagged words are indicated from within a text editor (EMACS in this case), and the user can choose between four operations for each element of the error list:

[i] Replace this particular occurrence of the word with a correction suggested by the user (the word was wrong in this context).

[ii] Replace this word globally (throughout the text) by a correction suggested by the user (the word is wrong in any context).

[iii] Do not change the word, and do not add it to the user-defined lexicon (the word is normally wrong, but not in this context).

[iv] Do not change the word, and add it with all its derived forms to the user-defined lexicon (the word is correct in any context).

Notice that in options [i] and [ii], an iterative verification of the input by the user is necessary.

We have implemented some further features to make life easier for users of the system. These include a symbol which can be inserted in the source text. It notifies the system that the word form following it is correct and should not be looked up but added to the user-defined lexicon with all its derived forms. This symbol can be used in front of words which the user suspects to be missing from the dictionary system but are nevertheless correct. Another symbol makes DSPELL skip a specified number of lines. This way DSPELL can be forced to jump over non-Dutch text fragments, bibliographies, program fragments, etc. Finally, with each run of the program, some statistical information about the source text is provided: number of words, the most frequent content words (a primitive index), number of presumed er-

[12] Some 500 words would cover 60% of word occurrences in English text, and 16,000 words 95% (Data by Greanias, 1984). He does not specify whether 'words' refers to word forms or to paradigms.

rors, number of words found in each dictionary, type-token ratio, number of words morphologically analysed, etc. Some of this information can be used in the computation of readability scores.

Figure 1 shows the flow of control in the *detection part* of the algorithm.

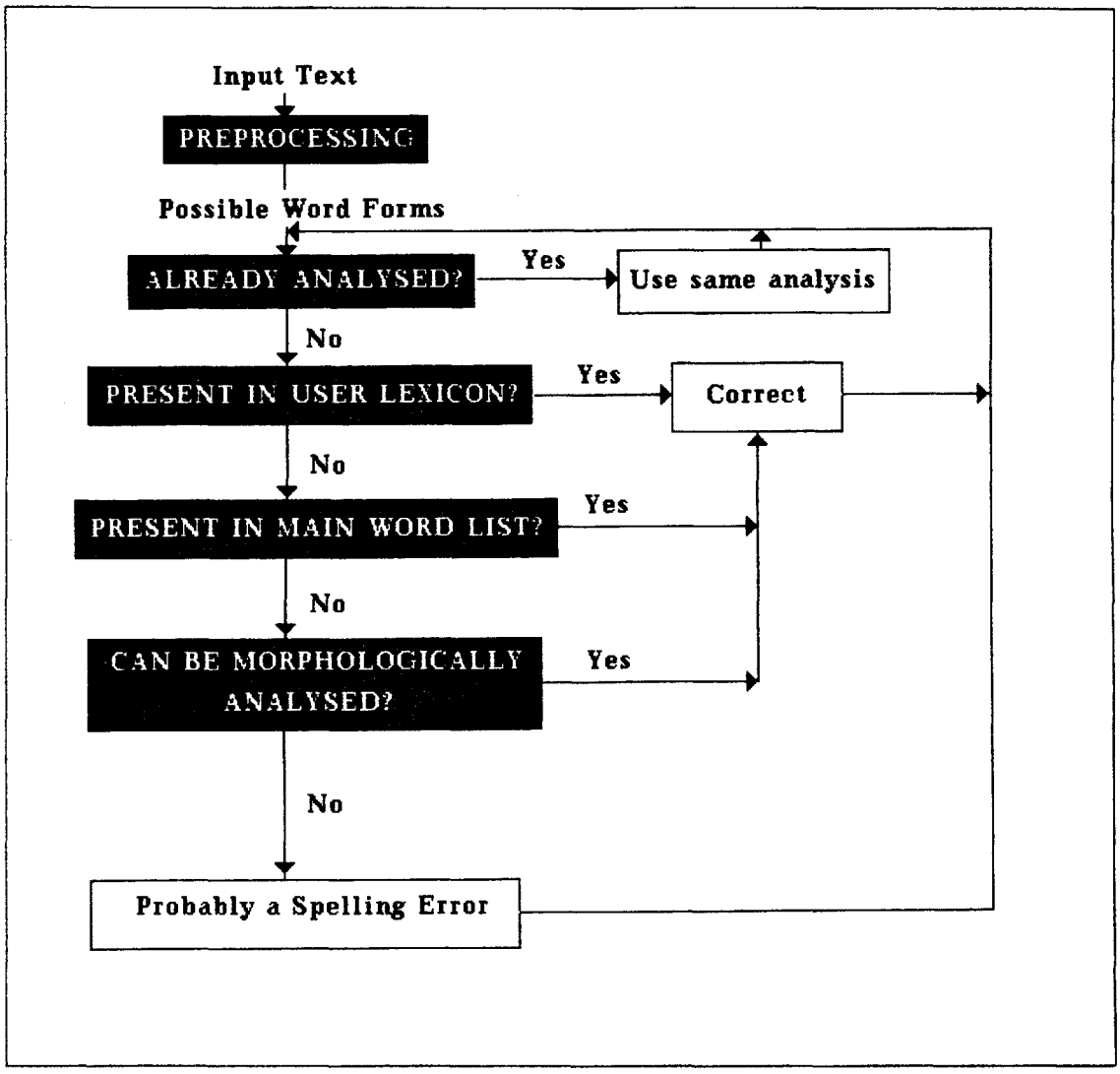


Figure 1. Flow of control in the detection part of the algorithm.

6.2.2 Evaluation of the Program

A reasonable performance evaluation of spelling checkers would have to take into account processing speed and an *error percentage* such as (1).

(1) $(O + U)/2$ where $O = o/c$ and $U = u/e$

Here, c is the number of correct word types[13] in the text, o the overkill (false alarms: number of word types incorrectly indicated as an error), e the real number of errors in the text, and u the undershoot (missed hits: number of errors not found). Our formula regards undershoot as worse than overkill, because U (which relates the number of undershoot errors to the total number of spelling errors) increases much faster than O (which relates the number of overkill errors to the number of word types in the text). This can be explained by the fact that there are more word types than errors in a text. Therefore, one spelling error missed weighs heavier than one correct word flagged. This was done intentionally as it reflects our opinion about the relative importance of both types of errors. Ignoring an incorrectly flagged word is easier than finding a missed error. The two error rates O and U can also be evaluated independently.

Most published accounts of evaluation measures for spelling checkers tend to ignore overkill errors, or they work with word tokens instead of types, or measure undershoot in relation to the number of words in the text instead of the number of errors actually made. Furthermore, accounts of time may refer to both real time and processing time. Both depend on the type of processor which is used to run the program. All these facts make it difficult to compare different systems[14].

In the absence of comparable quantitative information, we refrained from testing the performance of DSPELL by computing the error percentage described above on sample texts. However, we did test the reduction of overshoot obtained by the incorporation of a morphological parser in our algorithm. DSPELL was applied to a randomly chosen text containing 942 word tokens (399 types). Without morphological analysis the overshoot ratio (percentage of word types incorrectly indicated as an error) equaled 12%. With morphological analysis included, this error percentage was drastically reduced to only 2% (which is a far more acceptable ratio). Forty correct words which would have been flagged by a normal checker were accepted without

[13] We use types here instead of tokens because it is easy to introduce a facility to prevent the user from being confronted more than once with a correct word which is flagged. See the program description above.

[14] It would be interesting to make a 'benchmark' comparison of different spelling checkers (implemented in the same computer language, running on the same machine) for Dutch. To our knowledge, no commercially available spelling checkers developed especially for Dutch exist yet, but various firms (a.o. Wang, OCE and IBM) seem to be working on it.

problem by DSPELL.

We will conclude this section on *error detection* with a qualitative comparison of our approach to other systems. We will discuss two alternative approaches: trigram analysis and pattern recognition.

The philosophy behind *trigram analysis* is that a text or a set of texts can be characterised by the letter strings which occur in them (Chudacek, 1983; Chudacek and Benschop, 1981; De Heer, 1982). New texts in the set, or new words in the text, should comform to this characterisation. Considering an alphabet of 27 characters (26 letters and a blank), 19,683 trigrams (letter strings of length three)[15] are possible. In Dutch, only 10 to 20 percent of these are possible in principle. If we store a table of these trigrams in memory (possibly with their frequency), we can proceed as follows. We analyse each word of the input text into trigrams (its 'syntactic trace'). E.g. *error* becomes {^er err rro ror or^} (^ stands for a blank here). Whenever a trigram is found which is not in the table or the probability of which does not reach an empirically determined threshold, the word is probably an error. This approach was used in the SPEEDCOP project of Chemical Abstracts Service (Zamora, Pollock and Zamora, 1981; Zamora, 1980). For several chemical databases, a characterising set of trigrams was computed, and used in error detection. There are two important drawbacks to this method: for each corpus a new set of trigrams should be computed, and the results were poor, with value pairs for U and O between 95/32 percent and 32/95.5 percent, depending on the setting of a trigram frequency threshold. Our own experiments point in the same direction: discrimination between correct and false words is not possible with trigram analysis alone[16]; a trigram table sufficiently large to deal with the problem of overkill cannot handle undershoot satisfactorily, and vice versa. Nevertheless, trigram analysis combined with a dictionary lookup approach can boost the performance of the program by restricting the number of dictionary lookups needed.

---

[15] There is no theoretical reason to prefer trigrams to digrams or n-grams of an order higher than three. The choice is based on practical considerations (available memory, search speed) and empirical comparisons, which made clear that digram sets are not characteristic enough (Chudacek and Benschop, 1981) and that, with higher order n-grams, the difference between n-gram analysis and dictionary lookup fades. The latter has been described as n-gram analysis with variable n (Zamora, Pollock and Zamora, 1981).

[16] Notice that at this point, we are saying nothing about the usability of a trigram approach for the suggestion of *corrections* for errors.

Boot's *pattern recognition* approach to hyphenation (discussed in section 5.2) has also been suggested as useful for spelling error detection (Boot, 1984). This strategy consists of checking the well-formedness of word-initial and word-final consonant clusters. Boot sees no solution for word-internal consonant clusters. Obviously, compound analysis is necessary to solve this problem. As the word form parser described in Chapter 3 and used in the detection program makes use of morpheme structure conditions, and as word-internal clusters are checked in the process of morphological analysis, the DSPELL program described earlier is clearly superior to the suggestion by Boot. In essence, morpheme structure conditions are a subset of Markov transitional probabilities (namely, transitions with either 1 or 0 probability). Markov chains have also been applied to error correction (Srihari, Hull and Choudhari, 1983) but have proved to be insufficient if not combined with dictionary lookup.

6.3 Correction

The remainder of this chapter is of a more theoretical nature. We have not implemented a complete correction algorithm, but we implemented parts of both similarity measures and error transformations in order to be able to support our theoretical exposition.

Whenever a string of characters is flagged as an error, an artificial corrector should be able to suggest corrections for it. Two strategies are traditionally used:

[i] Generate, by means of a set of transformations on the alleged error, all possible words which may have been intended by the author (an *error grammar approach*, e.g. Yannakoudakis and Fawthrop, 1983a and 1983b; Peterson, 1980a and 1980b).

[ii] Compare the alleged error to each dictionary entry, using a similarity measure. The dictionary items most similar to the error are possible corrections (a *similarity measure approach*, e.g. Angell, Freund and Willett, 1983).

We will discuss traditional approaches of both types, and a new one based on phonemisation which can be adapted to belong to either type. Furthermore, a heuristic solution is provided for the problem of spelling errors within compounds which are not listed in the dictionary. Some expected hardware developments which may make fast on-line correction a nearby possibility are cited; and finally, we take up the problem of performance evaluation again.

6.3.1 The Error Grammar Model

In a classic study, Damerau (1964) argued that some 80 percent[17] of all word level errors belongs to one of the following formal categories:

[i] *Deletion* (one letter missing: *camoufage*)

[ii] *Insertion* (one letter too much: *camoufflage*)

[iii] *Substitution* (one letter wrong: *camouflahe*)

[iv] *Transposition* (two adjacent letters transposed: *camoulfage*)

Complex errors (combinations of the four categories mentioned) would account for the other 20 percent. The error grammar approach in its simplest form would be to generate all substitutions, insertions, deletions and transpositions of a suspected error, and look them up in the dictionary[18]. Figure 2 gives an example: *appel* (apple) is misspelled *apel*

Input	Generated Forms	Forms Found In Lexicon
apel	pael, aepl, aple	
	pel, ael, apl, ape	pel
	?apel, a?pel, ap?el, ape?l,	papel, kapel, lapel,
	apel?	appel, ampel, aprel
	?pel, a?el, ap?l, ape?	abel, adel, agel, avel,
		awel, apex

Figure 2: Error Grammar Approach to the correction of *apel*.

In the example, a question mark stands for any letter in the alphabet. All forms generated (241 in this case) were looked up in a dictionary (a computer version of the Van Dale dictionary; Kruyskamp, 1982). The number of generated forms (which means the number of lookups necessary) can be calculated with formula (2),

(2) $$k(2n + 1) + 2n - 1$$

[17] Ninety to 95 percent in other sources (Greanias, 1984).

[18] Note that in a system with unlimited resources, this technique can, theoretically, be reversed to prevent undershoot: if all transpositions, substitutions, and deletions of all words in the dictionary were stored, we could immediately check from which correct words another formally correct word might be a misspelling. Only when this is not the case, we can be relatively sure that the word is correct. A dictionary expanded this way would contain some 100 million entries (probably a fraction of this amount if morpheme structure conditions are taken into account).

where *k* is the number of letters in the alphabet and *n* the length of the word in letters. The long list of suggested corrections (generated forms present in the dictionary) confronts us with an important shortcoming of this method: such a high amount of possible corrections is a burden rather than a relief to the user of the system. Especially with short words and large lexical databases[19], the problem is acute. Another drawback is the processing speed (or rather the lack of it). In short, the error grammar approach is terribly inefficient[20]. It seems clear that we need ways of (a) constraining the number of 'hypotheses' generated by the error grammar, and (b) restricting the number of hypotheses accepted after dictionary lookup. To accomplish (a), we could begin with refining the rules of the error grammar. It would be natural to start looking for some 'systematicity in failure'. We will list here some empirical data about typing and spelling errors (Shaffer and Hardwick, 1969; Yannakoudakis and Fawthrop, 1983a; Van Nes, 1976; Glencross, Courner and Nilsson, 1979; Grudin, 1983).

Spelling Errors.

(i) Relatively few errors are made in the first letter of a word.

(ii) Doubling and singling of a letter (special cases of insertion/deletion) are common.

(iii) Some consonants are more often substituted than others.

More detailed rules of spelling errors can be found by studying a corpus of spelling errors, as was done by Yannakoudakis and Fawthrop (1983a) for English. However, these rules, unlike those mentioned in (i-iii), are language-specific (they depend on the phoneme/grapheme relationships which hold in a particular language).

[19] This need not surprise us, for it is basically the same issue as the undershoot problem in detection discussed earlier: what we do in an error grammar approach, is *mutatis mutandis* comparable to generating errors, and a large dictionary increases the probability that such an error is present in it.

[20] The error grammar approach may well be more efficient in logic programming. Berghel and Traudt (1986) describe a Prolog implementation of spelling error correction through checking the transformations specified by Damerau (1964). In Prolog, instead of a generate + test sequence for each transformation, the dictionary is searched with a string containing a variable, collecting all forms that match this string. The unification matching necessary to do this is built in in Prolog. This leads to an elegant program, but we have our doubts about the efficiency and feasibility of the approach in a practical application: looking up k*n substitutions is less costly than comparing n strings with one variable to all dictionary items, if the size of the dictionary is large. Furthermore, as in Prolog data and programs are not distinguished, the complete dictionary must be kept in internal memory, resulting in a serious overhead when the dictionary is of a normal size.

For Dutch, the following rules can be added (based on Booij, Hamans, Verhoeven, Balk and Van Minnen, 1979).

(i) Confusion between *t*, *d* and *dt* in verb forms (*Dt*-mistakes: see later this chapter and Chapter 7) is frequent.

(ii) Confusion between *ei* and *ij*. The difference is only etymologically motivated.

(iii) Spelling of loan words. A lot of confusion in this matter arises from the fact that two partially conflicting spelling systems exist for Dutch. One is 'permitted', the other is 'preferred'.

(iv) Grapheme for voiced obstruent instead of voiceless counterpart and vice versa (*f,v;s,z;g,ch*).

(v) Omission or hypercorrect insertion of a word-final <n>.

Typing Errors.

(i) Most typing errors (from 50 to 80 percent) are due to hitting a key in the neighbourhood of the target key, resulting in a substitution error.

(ii) The probability of an error increases with increasing word length and diminishing word structure.

(iii) Many typing errors (between 10 and 76 percent, depending on the source) are discovered by the typist him/herself.[21]

(iv) Transposition (mostly across hands) is another important error category.

(v) The difference in length between the correct form and the erroneous form is mostly one character, and rarely more than two characters. The error form is mostly shorter than the correct form. This goes for spelling errors as well.

(vi) There is a distinct frequency effect in substitution typing errors: the most frequent letter is almost without exception substituted for the less frequent.

Systematicity in typing errors does not depend on language, but on keyboard and typist. However, the influence of the former is greater than the influence of the latter.

These data can be used in a straightforward way to constrain the number of hypotheses generated by an error grammar. Using them, we can in a sense predict

[21] Though not really relevant in the constraining of the error grammar, this observation is important as it supports to some extent the position that automatic typing error correction is not useful. But see section 6.4.

which errors and corrections are most probable.[22] However, most experimental data are based on English, and, even for that language, they are incomplete. Reliable additional data for Dutch are badly needed. As regards the restriction of the number of hypotheses accepted after dictionary lookup, the same rules, which are used to constrain the generation of hypotheses, can also serve to assign a probability to the hypotheses found in the dictionary. A probability threshold can then be used to select one or more possible corrections. This can also be done by keeping the dictionary to a reasonable size, and by using sentence-level information (syntactic structure) to reject suggested corrections. A simple surface parsing could exclude hypotheses which belong to particular (sub)categories because they do not fit in the syntactic structure.

6.3.2 The Similarity Measure Model

In this approach, a detected error is compared to all entries of a dictionary, using a similarity measure. The dictionary entries most similar to the error are considered possible corrections. One useful implementation of this approach is the DWIM (Do What I Mean) facility of the programming language Interlisp (Teitelman, 1978). A detected error (an identifier unknown to the Lisp interpreter) is compared with words in a spelling list (dictionary). The similarity between the suspected error and an element from the spelling list is defined as inversely proportional to the number of differences between them (based on a letter-by-letter comparison), and proportional to the length of the longest word. Transposition and doubling are not counted as differences, and if an element on the spelling list is much longer or shorter than the error, it is immediately rejected as a correction. The system works perfectly in applications with a short word list such as the DWIM facility (Lisp function and variable names), but it becomes expensive (in processing time) and error-prone, if a large lexical database must be used. Again we can use the experimental data listed earlier to constrain the similarity search; e.g., we could exclude from comparison those dictionary entries which do not begin with the same letter as the error, and those that differ more than two letters in length. But this increases the possibility that errors or incompleteness are introduced.

Several methods exist to compute the similarity between two strings (E.g., see Faulk, 1964): number of identical letters (material similarity), number of letters in

[22] Attractive as this may seem, being a heuristic and not an algorithm, this solution brings with it a probability of failure.

the same order (ordinal similarity) and number of letters in the same place (positional similarity). Blends are always possible. For example the computation of the intersection of the syntactic trace (the set of trigrams) of two strings boils down to computing material similarity while taking into account ordinal similarity as well (Angell, Freund and Willett, 1983). Sometimes abbreviations are made of the letter strings before comparing them, as in the SOUNDEX system (e.g. Joseph and Wong, 1979) and variants thereof (Boot, 1984). A typical abbreviation would be the first letter of a word, followed by three numerals representing a class of related consonants (e.g. *m and n*) in their order of occurrence (of double consonants, only one is kept).

In a limited, explorative study, we tested some of these measures on fifty word strings with one error in each of them. The strings varied in length, in the kind of error present, and in the position of the error in the string. Each string was compared to its error-less counterpart, using five different similarity measures: *material, positional, trigram overlap, ordinal-1* (an abbreviation combined with positional similarity measure) and *ordinal-2* (an abbreviation combined with material similarity). The results are presented in Tables 2, 3, and 4.

	String Length		
	Five	Eight	Twelve
Material	85.5	91.0	93.8
Positional	61.1	65.0	71.0
Ordinal-1	74.8	72.5	85.0
Ordinal-2	78.6	88.8	95.0
Trigrams	53.1	65.4	77.3

Table 3. Similarity between error form and correct form with different similarity measures relative to string length (values are mean percentages).

	Position of error in string		
	Beginning	Middle	End
Material	90.2	90.2	90.2
Positional	49.1	66.4	81.6
Ordinal-1	65.0	78.8	88.6
Ordinal-2	85.0	88.8	88.6
Trigrams	65.7	60.7	69.5

Table 3. Similarity between error form and correct form with different similarity measures, relative to position of error in string (values are mean percentages).

	Error Type			
	Transposition	Substitution	Insertion	Deletion
Material	100.0	86.4	88.1	86.4
Positional	72.8	96.4	49.9	53.8
Ordinal-1	90.7	61.5	66.8	90.7
Ordinal-2	96.3	77.6	82.4	93.5
Trigrams	50.9	65.0	76.3	69.0

Table 3. Similarity between error form and correct form for different similarity measures, relative to kind of error (values are mean percentages).

Similarity tended to be higher with increasing word length for all types of similarity measure. Material similarity yielded consistent results for all types of error transformation, and was insensitive to the place of the error in the string. But two strings which are anagrams of each other are equal in this approach. Positional similarity performed badly whenever an insertion, deletion or transposition occurred at the beginning of a word. In fact it was only suited to correct substitution errors. The performance of ordinal similarity proved unreliable when combined with positional similarity. But, when combined with material similarity, it was a serious rival to material similarity alone. A trigram overlap similarity measure performed badly with transposition errors. In conclusion, of all similarity measures checked, material similarity came out best.

6.3.3 The Phonemisation Model

The rationale behind this model is the expectation that people will be confused and likely to make mistakes when spelling and pronunciation differ. Whenever they are unsure about the spelling image of a word, they will write what they hear (phonological or phonetic spelling). Therefore, we could develop a system that transliterates the erroneous spelling form into a phonological representation. We can use the phonemisation algorithm of section 4.2 to do this. This phonological representation is looked up in a dictionary with a phoneme representation as entry and spelling(s) as dictionary information.

Some frequent systematic spelling errors (as listed in Booij et al., 1979): e.g. double versus single consonants and vowels, omission or hypercorrect insertion of a final <n>, confusion about (de)voicing of <z>, <v>, <s>, and <f>, can be detected and corrected by this model. As such, the approach could be useful in a computer-assisted instruction environment, if it is extended with an algorithm to

detect and correct *dt*-mistakes[23]. The latter type of mistake cannot always be corrected or even detected in a phonemisation approach. Figure 3 shows some examples.

Input	Phoneme Representation	Alternative Spellings after Dictionary Lookup
vermoort	/vərmort/	vermoord, vermoordt
beil	/bɛɪl/	bijl
lope	/lopə/	lopen
lige	/lIγə/	liggen

Figure 3. Some examples of a phonemisation approach to correction.

We believe that this architecture has a limited use in general spelling error correction; typing errors can (in general) not be corrected by it, and the problem of multiple corrections suggested by the system remains. It can however be applied in some applications involving a restricted vocabulary or in combination with an algorithm for typing error correction.

A first attempt at such a correction system (for the correction of proper names) was made at the university of Nijmegen (Van Berkel, 1986). Her system involves the transformation of each (spelled) dictionary item to a network relating all possible pronunciations. E.g. *chrysant* (chrysantemum) is stored as the network ((X k) r (I y) (s z) Ant). This transformation is based on the phonological knowledge base and the syllabification and phonemisation algorithms discussed in Chapter 4. A suspected error is transformed into a phonological code via the same algorithm, and the resulting pattern is matched to the networks in the dictionary. This system is roughly comparable to a *similarity measure* approach.

A variant of the *error grammar* approach which should be considered in this context, is an algorithm that (a) transliterates the error into a phoneme representation, (b) generates different spelling images by a reverse application of the rules in Chapter 4. E.g. phoneme /ə/ can be realised as (among others) <e>, <i> or <ij> in

[23] Confusion between <t>, <d> and <dt> in the formation of present participle, past participle and singular simple present forms of verbs. Number and person of the subject of the sentence determine the form of the verb. E.g. *ik word* (I become), *je wordt* (you become), *word* je (do you become; inversion), hij *wordt* (he becomes), etc. The problem derives from the fact that the pronunciation of the highlighted word endings is always /t/. See also Chapter 7.

spelling, these possibilities should all be generated, and (c) looks up the generated forms in a normal spelling dictionary. Those which exist are possible corrections. It remains to be seen if such a system can avoid a combinatorial explosion of computed spelling images, which would prevent its being used in real time.

A promising new variant of the phonemisation approach, announced in Van Berkel (1986), is the combination of phonemisation with trigram analysis (or rather tri*phone* analysis). In this variant the following steps are taken:

(i) All words in the lexicon are transformed into one or more phoneme representations.

(ii) The phoneme representations are transformed into sets of triphones.

(iii) Each triphone contains pointers to all spelling forms of which the phoneme representation(s) contain(s) this triphone (i.e., an inverted file is created). Each triphone is assigned an information value, relative to the number of pointers it contains.

(iv) Of an input form to be corrected, the phoneme representation(s) are first computed, and then the associated triphone representations. Those triphones which are most selective (i.e. have the highest information value), are used to collect possible candidates for the correction of the input form (by using the inverted file). The candidate most similar to the input form (using a similarity measure), is then selected.

The most important advantage of this approach is that it (theoretically) allows spelling errors and typing errors to be corrected by a single algorithm.

6.3.4 New Hardware

Whichever method is applied, correction is prohibitively expensive in processing time. New hardware may provide a solution to this problem. Recently, a new chip (PF-474) which can compare strings of letters or words, and compute a similarity measure was announced by Proxemics Technology (Yianilos, 1983). It works by purely statistical methods, but a phonetic or orthographic similarity can be simulated by chosing a proper alphabet, and by programming the three parameters determining the similarity measure differently for each element of the alphabet. It keeps in store sixteen pairs with the highest similarity-value for later processing. The main advantage of this new chip is its speed: the relation between string length and processing time is linear instead of exponential. Furthermore, several PF-474s can be arranged

in parallel, which makes the searching of large databases in fractions of seconds possible.

Another useful hardware development to be expected soon, is the availability of large dictionaries on optical disk. The fact that these storage media are *read-only* is not really a disadvantage, because updating can be delegated to a modifiable user-defined lexicon which acts as a shell around the main dictionary on disk. Dictionaries stored in 'firmware' may also be helpful to improve storage capacity and decrease access time.

The value of these hardware developments lies in the fact that the introduction of heuristics, of which the main purpose is to save search time and memory space, might become superfluous. These heuristics often introduce a probability of errors. An algorithmic, complete solution should always be preferred to a heuristic solution, if the former is feasible in terms of efficiency.

6.3.5 A Note on the Correction of Spelling Errors in Compounds

Misspelled compounds the correct forms of which are not present in the dictionary confront us with a problem. Consider a compound which is not present in the dictionary, with an error in one of its parts; e.g.: *onderwjisprogramma* (onderwijsprogramma, education program; a transposition error). This string is correctly flagged as an error by the detection module, but correction is not possible with a similarity measure or an error grammar approach as these algorithms are based on comparison of the complete string with all or a subset of the dictionary entries and on lookup of the complete string in the dictionary, respectively. The most straightforward method to overcome this problem in an error grammar approach, would be to include morphological analysis in the process of looking up suggested corrections, but this would mean an explosive increase in the number of dictionary lookups as analysis presupposes a large number of lookup actions.

Another possible solution — this time in the context of a similarity measure approach — would be to resort to an algorithm in which morphological analysis and correction are intertwined. This integration of analysis and correction can be achieved by using the similarity measure method as a general dictionary lookup procedure in the detection phase. During morphological analysis, each dictionary lookup produces a similarity measure. If the similarity measure denotes equality, a 'normal' dictionary lookup occurs; if an identical string is not present in the dictionary, those

dictionary items of which the similarity measure exceeds an empirically determined threshold are selected as corrections of the original string. This method is extremely expensive since all dictionary items must be checked each time a dictionary lookup is necessary, which may be many times for each word morphologically analysed. In fact, both methods proposed up to now are only feasible if search is sufficiently fast (based on a hardware implementation of the search algorithm, preferably in a parallel architecture) and if the number of selected dictionary items for each lookup is sufficiently low. Otherwise, the parsing part of the algorithm would have to deal with too many alternative segmentations, and the probability of false parses would increase.

Yet another alternative possibility involves a special segmentation procedure during analysis. This procedure consists of a relaxation of the constraint that the subparts found in a string should fill this string completely.[24] In short: holes would be allowed in the segmentation. E.g.: *onder#xxxx#programma*. This 'hole' could then be corrected by one of the correction methods described earlier (with error grammar or similarity measure). In practice this approach leads to many problems as well. The example above would be treated as *onder#xx#is#programma*, leading to an unnecessary correction attempt of *wj*. The solution (a heuristic, rather) is to analyse a string for either the largest left or the largest right part found in the dictionary, and to make the rest a hole. In the example: *xxxxxxxxx#programma* (largest right part, because it is larger than the largest left part, which is *onder*). The probability that the hole is listed in the dictionary is sufficiently high, because most compounds consist of two parts, which are either simplex words or compounds listed in the dictionary.

An additional advantage of this approach is that the number of suggested corrections can be constrained during the parsing phase of the analysis. The parser can produce information about the possible parts of speech of the 'hole'. Only words in the dictionary which belong to these categories have to be checked during correction. A short overview of our adaptation of the two stage correction algorithm is given in Figure 4.

---

[24] A similar relaxation in the parsing part of analysis, allowing combinations of segmented word forms normally not allowed by the compound grammar, would make possible the correction of errors resulting from the accidental concatenation of different words. E.g. *He worked veryhard.* Unfortunately, it is not evident under which conditions the constraint should be relaxed.

```
1. DETECTION, which consists of:
  a. Dictionary lookup of the complete string (also in user-defined
     dictionary and document dictionary)
  b. (if not found) Morphological Analysis, which consists of:
       i. Segmentation-1 (find all possible morpheme distributions),
          which implies a number of dictionary lookups.
      ii. Segmentation-2 (if segmentation-1 fails);
          look for largest dictionary entry in the string, starting from
          either left or right. Declare the rest a hole.
     iii. Parsing, keep those distributions that are allowed by the
          morphological grammar, or signal error if no analysis is found.
          If a hole is present, determine the possible parts of speech
          for it.
2. (If error is signaled or hole is detected) CORRECTION,
       Either:
  a. Generate all possible modifications by means of
        (i)  an error grammar or
        (ii) a phonemisation algorithm,
     and look them up in the dictionary (those that are found are
     corrections).
       Or:
  b. Compare the flagged string or hole to all dictionary items using a
     similarity measure. In the case of holes, dictionary lookup is
     constrained by part-of-speech information.
```

Figure 4. Adapted two-stage model for word level correction.

In the example, *onderwjisprogramma* would be input, and not found in the dictionary (1.a). During morphological analysis, segmentation strategy One fails (1.b.i) which implies that a hole is created; in this case *xxxxxxxxx#programma* (1.b.ii). During parsing (1.b.iii), it is determined that the hole can be N, V or A. In the correction part, either an error grammar approach (generate modifications of *onderwjis* and look them up; 2.a) or a similarity measure can be used to correct *onderwjis*. In 2, only dictionary entries with category N, V or A are considered.

6.3.6 Performance

The remarks we made about performance of spelling error detectors hold for correction as well: results cannot be compared easily. One has to be careful with performance descriptions. For instance, Angell, Freund and Willett (1983) do not count undershoot errors of their program, nor errors due to the incompleteness of their dictionary. When you do take these into account, the success-rates decrease from 75 percent and 90 percent to 67 percent and 84 percent, respectively. Similarly, Yannakoudakis and Fawthrop (1983b), dismiss 'cases where the algorithm searched the wrong part of the dictionary, and cases where the error form was not in the dictionary' as unimportant. When considering these cases important, the success rate drops from 90 to 68 percent. Even more doubtful is the fact that they tested the perfor-

mance of the program on data (misspellings) which they used to extract the rules on which their algorithm is based. Only 10 percent of the data used was new. The high performance of their algorithm is therefore not surprising. Performance on the new data alone is only 62 percent. The performance of a correction program should be judged from the percentage of alleged spelling errors it can correct, the processing time needed to do this, and the mean number of possible corrections it suggests.

6.4 Conclusion

We continued our presentation of a knowledge-based approach to language technology. The linguistic knowledge in Part II (morphological analysis and phonemisation) was proved to be indispensable for the development of high-quality spelling and typing error checkers and correctors. We presented DSPELL, a spelling checker for Dutch which takes into account the morphological structure of word forms, thereby reducing the rate of overkill errors with 10 percent.

As regards automatic correction, it will have become clear that no single strategy can handle the correction of all word-level mistakes. Typing errors and spelling errors share some features, but are different when we try to describe their systematicity in a set of rules. For some spelling errors such as *dt*-mistakes, special correction routines can be developed. Other spelling errors (e.g. doubling and singling of consonants and vowels) can be corrected with correction procedures developed for typing errors. The main problem is that both kinds of errors are usually present in the same text. The approach taken is also dependent on hardware possibilities and on the size of the dictionary. A similarity measure can only be used with small dictionaries. An integrated approach to word-level correction would take into account *text characteristics* (the subject matter of the text might determine the set of dictionaries used), *channel characteristics* (spelling errors, typing errors and transmission errors have a different systematicity), and *author characteristics* (programs which remember the errors a specific user often makes can be developed).

Is spelling error correction useful? It could be argued (as was done by Prescott Loui, 1981) that, however sophisticated the program, humans would still be needed to proofread documents because there is always a small percentage of errors which is not detected (consistency, semantics, omitted words and paragraphs, undershoot). Therefore, automatic error detection is useless and even disadvantageous, because human correctors perform better when the number of errors is large (if an error is

found only sporadically, they will become bored or deluded, because there is no reward for their attention). We think this is a debatable position (even leaving aside the pseudo-psychological explanation of the behaviour of proofreaders), because spelling verification should be seen in the context of an Author Environment; an interactive environment in which writers *develop* text. Any help they can get at this stage to keep their mind from mundane typographical conventions and to prevent that they get stuck in grammar[25] will be time saved to work on the ideas they want to express.

[25] The phenomenon of getting stuck in syntactic and spelling matters while writing has been termed *downsliding* by Collins and Gentner (1980).

CHAPTER 7

Intelligent Tutoring Systems

7.1 Introduction

ICAI systems cannot be AI systems warmed over.
J.S. Brown

The principal aim of computer assisted instruction is (or should be) the individualisation of education. Learners should be able to acquire and practice knowledge and skills at their own pace and the knowledge presented should be adapted at all times to their current level of expertise. Computer programs can be helpful to achieve this. Rather than replacing ordinary class teaching (which is indispensable to acquire social skills), computer aided instruction should supplement and amplify the former.

In this chapter, traditional and Artificial Intelligence approaches to computer assisted instruction will be contrasted, and an example of the latter will be presented. A problematic aspect of Dutch spelling education (verbal inflections) was chosen as the domain of an Intelligent Tutoring System (ITS) which incorporates expert knowledge about the teaching domain (based on the model of morpho-phonology outlined in Part II of this dissertation), a module to present and practice domain knowledge, a module to automatically diagnose the mistakes a learner makes, and an interactive user interface.

7.2 CAI versus ITS

Traditional CAI programs (programmed instruction systems) consist of a series of three steps, presented in a linear fashion and repeated a number of times.

(i) First, a chunk of subject matter programmed by the teacher is presented to the learner. Languages specifically suited for this purpose (*author-languages*) exist. Advanced CAI systems may even automatically produce teaching material (examples) for a restricted number of domains (e.g. arithmetic).

(ii) Second, a number of tests (questions and exercises) on the subject matter imparted during the first step is presented.

(iii) Finally, the answers to the tests provided by the learner are analysed. This stage may invoke *corrective feedback*, i.e. repetition of teaching material adapted to the learner's response, omission of subject matter considered known, additional tests etc.

Major shortcomings of this approach are the absence of expert knowledge about the domain being taught (making flexible reactions by the system to input from the learner impossible) and the absence of the ability to make a sufficiently detailed diagnosis of the learner's problems, and to provide explanatory feedback. Finally, the burden of supplying the chunks of knowledge in step (i) is completely on the shoulders of the teacher or programmer (see also Kempen, Schotel & Pijls, 1984).

Intelligent Tutoring Systems (also *Intelligent Computer Assisted Instruction*), try to repair some of the shortcomings of CAI systems. Most existing systems are prototypical and experimental (see Sleeman and Brown, 1982 for descriptions of a number of systems, and Yazdani, 1986 for a recent overview), yet some kind of 'methodology', largely based on the work of John Anderson and collaborators (e.g. Anderson and Reiser, 1985) has already been established. An ITS system consists of at least the following two modules:

(i) A knowledge-based *domain expert*. This module can solve problems about the subject matter in a human-like way. This means that e.g. an arithmetic expert does not add or subtract by manipulating strings of binary numbers, but by simulating the rules followed by human beings having expertise in the task. These rules may correspond to both implicit ('tacit') and explicit knowledge (rules learned at school).

(ii) *Didactic module.* This module includes a number of sub-modules. E.g. a *model of the learner* (his level of knowledge about the domain, the rules he uses), a *diagnosis system* (to refine the model of the learner on the basis of answers to questions or exercises), a *bug catalogue* (a list of frequent mistakes to guide the diagnosis procedure, error expectancies), a *tutoring module* (knowledge about what knowledge to impart to a particular learner and how to do it) and an *exercise generator.* Different ITS systems can be distinguished by the presence or absence of (sub-)modules, or by the emphasis put on them.

7.3 TDTDT: An ITS for Dutch Conjugation

In the remainder of this chapter TDTDT (Tdtdt Diagnoses Trouble with DT) will be described. TDTDT is an experimental ITS which teaches an aspect of Dutch spelling in which a lot of effort (on the part of both teacher and learner) is traditionally invested without very much result: the spelling of the conjugated forms of verbs. These poor results are typical of Dutch grammar and spelling teaching in general (see Pijls, Daelemans & Kempen, 1987 for a discussion of possible reasons for this).

7.3.1 The Problem

Dutch spelling is guided by two main principles: the *phonological principle* (write a word as it is pronounced in standard Dutch) and the *morphological principle* (spell related forms the same way). The former treats spelling as a (very) broad phonetic transcription of sound, the latter results in a transcription which is closer to the alleged lexical representation of the word form (Van Heuven, 1978).

Unfortunately for learners of Dutch, these principles are partially conflicting. Especially in the case of some verbal inflectional endings, a lot of confusion arises (Figure 1). Generally, mistakes involve an improper application of the phonological principle instead of the morphological principle in the spelling of verbal inflectional endings. E.g. learners are often at a loss whether a conjugational ending sounding like /t/ is written <dt>, <d> or <t>. The correct solution involves the application of a number of syntactic, morphological, phonological and spelling rules. A first prerequisite for an ITS for this subject is therefore that it be able to apply these rules correctly.

	Phonological Transcription	Lexical Representation	Spelling	Gloss	Principle
1	/blɛ^{i}f/	blijv	blijf	I stay	Phon
2	/lat/	laad	laad	I load	Morph
3	/lat/	laad+t	laadt	he loads	Morph
4	/ladə/	laad#de	laadde	I loaded	Morph
5	/gəlɛ^{i}t/	ge#leid+d	geleid	led	Phon+Morph

Figure 1. Phonological transcription, lexical representation, spelling, gloss and governing principle in a few conjugations.

7.3.2 The Domain Expert

This module contains the necessary linguistic knowledge, implemented as a system of KRS concepts, to conjugate any Dutch verb[26]. The system was described in detail in Chapters 3 and 4. For the purpose of the present application, explanatory information was attached to various concepts, and a rule-history subject was added to the word form concept. This slot lists the different decisions taken and rules applied to produce a particular word form.

7.3.3 Presentation of Domain Knowledge

Different pedagogical methods have been devised and introduced in education to teach this difficult aspect of Dutch spelling (see Assink, 1984 for an overview). A recent development is the *algorithmic rule method* (Assink, 1983 and 1984; Van Peer, 1982), which seems to have favourable results. In this method, an easily manipulatable *algorithm* is devised, pictured on a card, which allows quick and easy decisions on the spelling of verb forms. First, the learner is introduced to a number of concepts playing a role in the solution of the problem and featuring on the card (tense, finiteness, person, number etc.). Next, these concepts are related to spelling rules by means of the algorithmic decision scheme on the card. Exercises are made with the help of the card until the algorithm is internalised and the card becomes superfluous.

We have automated the second part of this teaching method (practicing the algorithmic decision scheme). We assume that learners working with our system have re-

[26] At present, syntactic knowledge (relevant for the controlling of subject-verb agreement) is absent in our system, but we are working on an integration with the syntactic parser developed by Konst (1986). An alternative we consider is to develop a special-purpose superficial syntactic parser for this problem.

ceived class room teaching about the different linguistic concepts used in the algorithm. For each specific conjugation problem, a decision tree is pictured on the screen (Figure 2). The terminology used in this picture does not necessarily coincide with the terminology presented to the learner. E.g. *Tense=Past?* could be presented as *did the action happen in the past* etc. The top part of Figure 2 shows the hierarchy of decisions to be taken. The shaded part represents the morphological rules to be applied and the spelling modifications to be made. We refer to Chapter 3 for a detailed account of these rules and modifications.

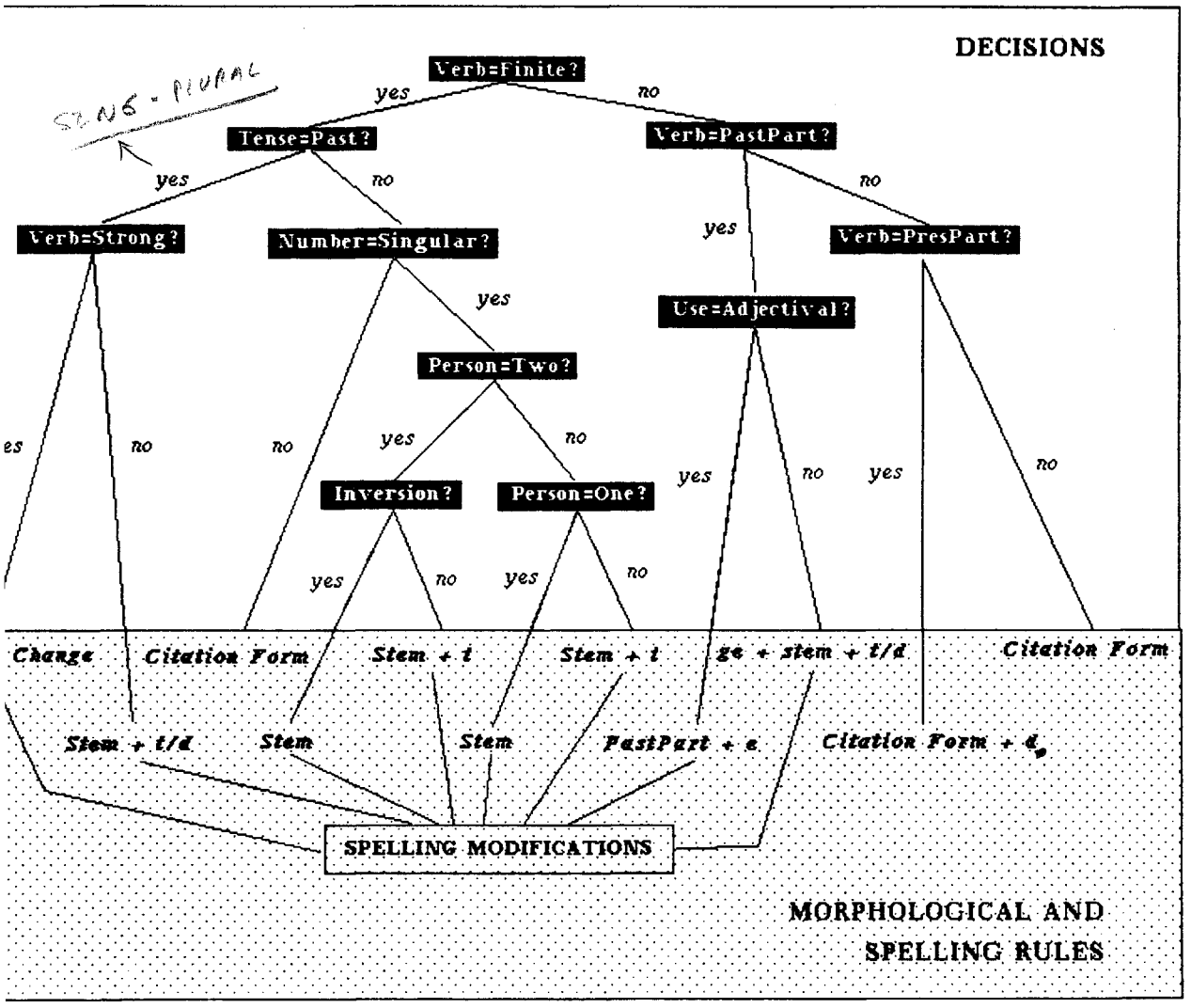


Figure 2. Decision tree for computation of verbal endings.

The solution of a specific conjugation problem is represented as a walk through the decision tree. Each node represents a question (e.g. *is it an action in the past?*) to which the learner answers by clicking on a menu item (see section 7.3.5 below). If a wrong choice is made this is explained to the learner. Teaching material (exercises)

This sequence fills the rule-history slot of the *redt* verb form after computation of this form by the domain expert.

Rules can be uniformly represented as consisting of a number of conditions and a number of actions, performed if the conditions apply in a particular context. The source of mistakes can now exhaustively be characterised as follows:

(i) At the macro-level: the omission or insertion of rules or the muddling up of the correct sequence of the rules.

(ii) At the micro-level: the omission of conditions or actions, the insertion of conditions or actions or the muddling up of the correct application order of conditions or actions. In our case, conditions are either decisions in the decision tree or conditions based on the form of the verb to be conjugated.

It will be clear from this that if we want to check all possible sources of an error, a combinatorial explosion of alternative possibilities results, even for relatively simple forms. Furthermore, the insertion of extraneous rules, conditions or actions is open-ended, and therefore unpredictable *in se*. This proves that an approach involving *exhaustive diagnosis* (like BUGGY for arithmetic; Brown and Burton, 1978) cannot be transported to more complex domains.

A solution to this problem commonly adopted is the compilation of a bug catalogue (e.g. Anderson and Reiser, 1985), based on an empirical study of the errors learners make, and on the relation of the results of this study to the different rules used by the domain expert. By means of the information thus acquired, it is possible to construct heuristic rules which guide the diagnosis process. However, compiling such a list is a time-consuming activity, and we believe that it can be automated to a large extent through progressive refinement of heuristic diagnosis rules acquired through interaction with learners.

Heuristics always treat a number of potentially relevant aspects of the problem (in this case diagnosis) as irrelevant. This can be justified if the number of things that actually go wrong in practice is considerably smaller than the number of things that can go wrong in principle. We believe this to be the case in the diagnosis of mistakes by learners in general and in the conjugation of verbs in particular.

We start from the rule history computed by the system. As a first hypothesis, the system assumes that the mistake is due to a wrong decision somewhere. All decisions in the list are negated in turn, and the effect of this on the computation of the

verb form is considered. Those decisions, which — when negated — result in the same mistake as was made by the learner, may possibly have caused the mistake. Note that the computation of the effect of the negation of a decision often involves the postulating of other decisions. E.g. negating past=no generates some new decisions such as weak=yes (or no). The system assumes the correct decisions to be taken. If several possible causes remain, the system tries to choose one by asking additional questions (E.g. *is it a weak verb?*) or by generating additional exercises. These additional questions and exercises are by no means unnatural to the learner because they take the same form as the 'normal' questions and exercises while interacting with the system.

The number of possibilities can then be constrained by computing the intersection (if the answer was again wrong) or the difference (if the answer was correct) of the relevant lists of decisions. If no hypotheses remain, i.e. all decisions taken by the learner are considered correct, the bug must be due to a mis-application of one of the rules. Roughly the same method can then be used to identify the rule in question.

From such a diagnostic interaction, a heuristic rule is derived. I.e. the association of a symptom (the wrong answer) with a diagnosis (a wrong decision or wrongly applied rule). Symptoms should be expressed at a suitable level of abstraction: it would not be very useful to have a specific verb as symptom. Rather the symptom is expressed in terms of the category of the verb (regular, semi-irregular or irregular classes) and the ending of the stem. In additional interactions with the same learner, already existing heuristic rules are gradually refined.

Our approach to rule-refinement is based on the approach to learning in second generation expert systems (Steels, 1985; Steels and Van de Velde, 1985). Second generation expert systems consist of a heuristic rule system (much the same as traditional expert systems) called the surface model, and an additional deep model which provides an understanding of the complete search space over which the heuristics operate. A second generation expert system can fall back on deep reasoning when its heuristic rules fail. The most important advantage of this architecture lies in the fact that it allows automatic learning of new heuristic rules. Rule learning happens by refinement (non-monotonically): a new heuristic rule is abstracted (in our case from an interaction with the learner), and is integrated into the already existing rule-base. This integration does not make the previous rules superfluous, but may restrict their application.

Two options are open in the use of heuristic rules: either a new rule base is constructed for each learner, or the same is used for all learners. In the latter case, the identity of the learner may be added to the list of symptoms.

7.3.5 User Interface

We used the full functionality of the Symbolics Lisp Machine mouse-menu-window system in the development of our interface to the learner. It has been shown (Schotel and Pijls, 1985) that children have no problem with such an interface and are motivated to work with it.

The decision tree is built up in interaction with the learner. At each node, a question is asked, and the answer (mostly yes or no) is given by clicking with the mouse on a menu of options. If the answer is correct, a branch and new node is drawn, and a new question is asked. If the answer is wrong, the learner is told so, and some explanation is given.

In an exercise (the computation of the correct form of a verb), the infinitive of a verb is presented in a syntactic context. The generating of these contexts is still a problem. Only limited information about the case frame associated with the verb is available from the lexical database, and no semantic information at all. Contexts are therefore very simple and canned phrases designed to be applicable to (almost) all verbs. The problem can be partly overcome by providing a more detailed and natural context and letting the learner choose an applicable verb himself (this limits his free choice, but not to a point where only one or a few possibilities are left open).

An alternative solution would be to explicitly ask for a specific form (e.g.: What is the inflected past participle of the verb *werken*, to work). In the latter case, a lot of the decisions that must be taken in the real-life computation of verb forms are given in the question, which diminishes the functionality of the tutoring environment considerably.

7.4 Conclusion

Intelligent Tutoring Systems are an important step in the liberation of class teaching from the less interesting aspects of the subject matter and in the adaptation of teaching to the needs and pace of individual learners. We have shown that an ITS can be developed even for subject matter traditionally considered 'complex'. It should be

emphasised that the system described relies on sophisticated hardware and software resources, presently out of reach for most schools. However, we are confident that the evolution of the microcomputer market will make possible the implementation of similar programs on cheap and small systems.

CHAPTER 8

Miscellaneous Applications

8.1 Rule Testing Devices

8.1.1 Background

One of the advantages of computer models of linguistic phenomena we mentioned in Chapter 1 is the framework they present for implementing, testing and evaluating linguistic theories. The construction of a working program forces the researcher to make all his assumptions explicit, and holes in his theory are inevitably brought to light. But there is also a more practical advantage. Developing a rule system which describes some kind of phenomenon is an intricate business. Small changes in rule-ordering or in conditions on rules can have enormous side-effects, and beyond a certain level of complexity the output of a set of interacting rules becomes unpredictable. It is therefore of considerable interest to the linguist to be able to test his rules and their interaction on a large amount of data. Apart from being a useful tool in debugging complex rule systems, a rule testing device can also be helpful in teaching linguistics because it makes visible the effect of the different rules and their interactions.

A system for developing and testing rule systems should conform to at least the following requirements: easy modification of rules should be possible, and traces of rule application should be made visible. The system described in the following section was developed with these goals in mind.

8.1.2 GRAFON[27]

GRAFON is a grapheme-to-phoneme transliteration and phonological rule testing system based on the syllabification and phonematisation algorithms described extensively in sections 4.1 and 4.2. In this chapter we will concentrate on the merits of the program as a tool in the development of phonological rule systems. The system takes words and sentence fragments as its input and transforms them into a phonological representation by applying a set of phonological and spelling rules. The level of phonological/phonetic detail can be adjusted[28] by adding or deleting rules.

The program does not take the form of a *rule compiler* for some specific formalism like generative phonology (see Fromkin and Rice, 1969; Kerkhoff, Wester and Boves, 1984 for examples of the compiler approach), as this would restrict the possibilities of the researcher to only that formalism. This implies, however, that the rules are implicit in the code of the program and that the researcher must be able to program in a sub-dialect of Lisp to use the system[29].

The program uses the three stage phonematisation algorithm described earlier (section 4.2):

(i) *Syllabification* (section 4.1) and *word stress assignment* (computation of syllable structure and retrieval of word stress)

(ii) *Transliteration* (transforming spelling syllables into phonological syllables)

(iii) *Phonological rule application.*

Stage (iii) is particularly important in rule testing. The functionality of GRAFON includes that phonological rules present in the system can be easily modified both at the macro level (reordering, removing and adding rules) and the micro level (adding, reordering and removing conditions and actions of rules). The domain (or scope) of a rule can be varied as well. Possible domains which can be

---

[27] This section is based on parts of Daelemans 1985b. The system is not to be confused with the GRAPHON system developed at the Technische Universität Wien (Founder and Kommenda, 1986) which is a grapheme-to-phoneme conversion system for German.

[28] Since it is not clear where exactly the boundary lies between a phonological and a phonetic representation, we will consistently use the term phonological here, even when the phenomenon described could be called phonetic.

[29] This approach is comparable to the the METAL translation system developed at LRC (University of Texas) and exploited by Siemens. In this system, too, the algorithmic and computational components are separated from the linguistic ones (Gebruers, 1986) but a minimal knowledge about Lisp functions is still necessary.

selected are syllable, morpheme, word and sentence. Furthermore, the application of various rules to an input text-string (its derivation) is automatically traced and can be made visible. For each phonological rule, GRAFON keeps a list of all input words to which the rule applies. This is advantageous when complex rule interactions must be studied.

Rules can be reordered and deleted by manipulating the elements of a simple rule list through menus. A new rule can be added by defining its conditions and the actions to be undertaken if the conditions apply. This can be achieved easily by using primitive Lisp functions like *and, or* and *not* in conjunction with predefined Lisp functions and predicates accessing the phonological data present in the system: E.g. (high-p x), (bilabial-p x), (syllabic-p x) etc. (see section 4.2.2). Another way the rules can make use of the phonological data is by means of simple self-explanatory transformation functions. E.g.: (make-voiced x), (make-short x) etc.. The precise manner in which the answer to these 'questions' and 'requests' is computed need not concern the user.

The motivation for this relatively independent interface to the phonological knowledge base is twofold: First, it allows us to model different theoretical formalisms using the same knowledge (e.g. the formalism of *generative phonology* can be modeled at the level of the user interface without having to modify the knowledge base). Second, the user is relieved from the burden of remembering how a particular piece of knowledge is represented (through default inheritance, computation, subject association or hierarchical relations). Furthermore, the flexibility in representing data (discussed in section 4.2.2) makes the system suitable as a tool box for researchers because the efficacy of different formalisms and categorisations can be easily compared.

The system is not exceedingly large or complicated, and the same architecture can be tuned to different natural languages and dialects. Due to its modularity, it can also be integrated into larger systems, e.g. morphological and syntactic analysis and synthesis systems, which opens up possibilities of studying rule interactions on a larger scale.

Apart from the fact that some knowledge about Lisp programming is necessary, there is one serious restriction on the system: no syntactic analysis is available at present, and therefore no intonation patterns and sentence stress can be computed. Moreover, it is impossible to experiment with the *phonological phrase* as a domain

for phonological rules. It is feasible in principle to integrate GRAFON with an object-oriented syntactic parser and with a program which computes intonation contours (Van Wijk and Kempen, 1985). If this were done, the phonological phrase, which restricts sandhi processes in Dutch, would become available as a domain for phonological rules, and the phoneme transcription of the input could be enriched with intonation contours.

On the next page, the environment in which a linguist working with GRAFON finds himself is pictured. The system has phonemised the phrase *halfvolle melk* (half-cream milk). The first line is the output of the syllabification algorithm. Stress is indicated by '+', internal word boundaries by '=', and external word boundaries by '==='. The derivation following the computed form lists for each syllable the rules which were applied to it. The central menu lists the options available within the GRAFON toplevel. *Stop* exits the toplevel and *Zuid* (south) and *Noord* (nord) select the rule-set associated with these variants of Dutch. The *Derivation* option prevents or causes the printing of a derivation for each computed form. Clicking with the mouse on *Rules*, causes the 'Phonological Rules' menu to appear (top right). This menu lists all rules known by the system. Clicking on the 'no' option after rules makes them inactive. That way, the overall effect of the presence or absence of a rule or set of rules can be examined. The *Examples* option makes the 'Give examples of:' menu appear (bottom right).[30] By clicking on the name of one of the rules the user gets a list of all inputs known by the system to which this rule was applied. Examples of hiatus-filling known to the system are printed bottom left of the picture. The examples obtained in a particular session with GRAFON can be stored in a file and loaded again during another session. Finally, the *Flush* option deletes all examples present in the system.

[30] Note that the picture is edited in the sense that in actual use, only one menu is present at the same time. We compressed three *bitmaps* of the Symbolics screen into one picture.

|'halə,fələ 'meləkǀ

"half"
(1 SCHWA-INSERTION)
(2 PROGRESIVE-ASSIMILATION)
(3 DEGEMINATION)
"melk"
(4 SCHWA-INSERTION)
>> m

Phonological Rules	yes	no
PROGRESSIVE-ASSIMILATION	☒	☐
REGRESSIVE-ASSIMILATION	☒	☐
INITIAL-DEVOICING	☒	☐
PLOSIVE-TO-FRICATIVE	☒	☐
NASAL-ASSIMILATION	☒	☐
DEGEMINATION	☒	☐
FINAL-DEVOICING	☒	☐
N-DELETION	☒	☐
PALATALISATION	☒	☐
HIATUS-FILLING	☒	☐
SCHWA-INSERTION	☒	☐
CLUSTER-REDUCTION	☒	☐
INTERVOCALIC-VOICING	☒	☐
VOWEL-DIPHTHONGISATION-1	☒	☐
VOWEL-DIPHTHONGISATION-2	☒	☐

Do It ☐ Abort ☐

Options within the GRAFON toplevel
Examples
Rules
Flush
Derivation
Zuid
Noord
Stop

Give examples of:
INITIAL-DEVOICING
FINAL-DEVOICING
PLOSIVE-TO-FRICATIVE
N-DELETION
PALATALISATION
HIATUS-FILLING ×
SCHWA-INSERTION
INTERVOCALIC-VOICING
VOWEL-DIPHTHONGISATION-1
VOWEL-DIPHTHONGISATION-2
PROGRESSIVE-ASSIMILATION
REGRESSIVE-ASSIMILATION
NASAL-ASSIMILATION
CLUSTER-REDUCTION
DEGEMINATION

Examples for rule #<HIATUS-FILLING 33743134>
rij in (+LO PEN == +DRIE == +RIJ EN == +TEE == +EL == +BUI ZEN)
rij in (+AAN == DE == +WIT TE == +WAND == +HAN GEN == +TWEE == SCHIL DE +RIJ EN)
zo in (+ZO +ALS == +DIE == +VRIEND == +DAT == +TROU WENS == +OOK == +AL TIJD == +HEEFT)
zo in (+ZO = +ALS == EN == PU +BLIE KE == +RUIM TE == +PAST)
po in (+WANT == PO E +ZIE == +KAN == MIS +SCHIEN == EN == +LEEG TE == +WAT == +VUL LEN)
rij in (DE == SCHIL DE +RIJ EN)
zo in (+ZO = +ALS == JE == +HEM == +HEBT == VER +LA TEN)

Lisp Listener 2

Some (verbal) descriptions of rules are presented in Appendix A.8, with some examples, a transcription computed by GRAFON, and the rule derivation as it would be presented to the user. The rules are listed in the order in which they are applied by the system. Alternative orderings with the same effect are possible. Not all restrictions and conditions are described for each rule. As the rules evolve continuously while experimenting with the system, the output should not be interpreted as a definitive position on phonological problems.

8.2 Automatic Dictionary Construction

8.2.1 Background

Although they play a central role in any working natural language processing system, lexical databases have been largely neglected in research. *Theoretical* accounts of parsing and generation mostly assume the existence of a lexicon with the relevant data, while the manual compilation of lexical databases (LDBs, machine-readable dictionaries) for *practical* applications still is an expensive and time-intensive drudgery. In the worst case, a LDB has to be built up from scratch, and even if one is available, it often does not come up to the requirements of a particular application.

We have developed a tool which helps both in the construction (extension and updating) of LDBs and in creating new LDBs on the basis of existing ones. Two levels of representation are distinguished: a static *storage level* and a dynamic *knowledge level*. The latter is an object-oriented environment containing linguistic and lexicographic knowledge. At this level, *constructors* and *filters* can be defined. Constructors are objects which extend the LDB both horizontally (new information) and vertically (new entries) using linguistic knowledge. Filters are objects which derive new LDBs from existing ones thereby optionally changing the storage structure. Filters use lexicographic knowledge. We will call a system coming up to these specifications a *Flexible Dictionary System*.

8.2.2 The Flexible Dictionary System

Levels of Representation. The main idea is to distinguish two representation levels: a static *storage level* and a dynamic *knowledge level*. At the storage level, lexical entries are represented simply as records (with fields for spelling, phoneme transcription, lexical representation, categorial information etc.) stored in text files for easy

portability. The knowledge level is an object-oriented environment, representing linguistic and lexicographical knowledge through a number of objects with attached information and procedures, organised in generalisation hierarchies (such as the KRS programs described in Part II). Records at the storage level are lexical objects in a 'frozen' state. When accessed from the knowledge level, these records 'come to life' as structured objects at some position in one or more generalisation hierarchies. In this process, record fields are interpreted as slot fillers. That way, a number of procedures and defaults becomes accessible (through inheritance) to these lexical objects. Figure 1 shows a static record and its associated KRS concept.

```
tafels              ta=fəl+s          102

(defconcept tafels
  (a noun-form
      (number [number plural])
      (stress [position 2])
      (spelling [string "tafels"])
      (lexical-representation [string "ta=fəl+s"])))
```

Figure 1. A static record structure and the corresponding KRS concept.

Constructors. For the creation and updating of dictionaries, *constructors* can be defined: objects at the knowledge level which compute new lexical objects (corresponding to new records at the storage level) and new information attached to already existing lexical objects (corresponding to new fields of existing records). To achieve this, constructor objects make use of information already present in the LDB and of primitive procedures attached to linguistic objects represented at the knowledge level. E.g. when a new citation form of a verb is entered at the knowledge level, constructors exist to compute the inflected forms of this form, the phonological transcription, syllable and morphological boundaries of the citation form and the inflected forms, and of the forms derived from these inflected forms, and so on recursively (see Part II). Our knowledge about Dutch morpho-phonology has not yet advanced to such a level of sophistication that an undebatable, exceptionless set of rules can be provided, making fully automatic extension of this kind possible. Therefore, the output of the constructors should be checked by the user. To this end, a cooperative *user interface* was built, reducing initiative from the user to a minimum. After checking by the user, newly created or modified lexical objects can be transformed again into 'frozen' records at the storage level.

Filters. *Filters* are another category of objects at the knowledge level. They use an existing dictionary to create a new one automatically (with filters, user intervention is not necessary). During this transformation, specified fields and entries are kept, and others are omitted. The storage strategy used may be changed as well. E.g. an indexed-sequential file of phoneme representations could be derived from a dictionary containing this as well as other information, and stored in another way (e.g. as a sequential text file). Filters use the lexicographic knowledge specified at the knowledge level to achieve this transformation. Lexicographic knowledge consists of a number of sorting routines and storage strategies (sequential, indexed-sequential, trees). We will call a lexical database obtained by means of a filter a *daughter dictionary* (DD) and the source a *mother dictionary* (MD). The MD incorporates our ideas about redundancy in lexical databases expressed in section 3.3; namely that current and forthcoming storage and search technology allow the construction, maintenance and use of large LDBs containing as much information as possible (see Byrd, 1983 for a similar argument). Constructors can be developed to assist in creating, extending and updating such a MD, thereby reducing its cost. Compact and efficient LDBs for specific applications or purposes can be derived from it by means of filters. The basic architecture of our Flexible Dictionary System is given in Figure 2.

Interactive User Interface. The aim of this interface was to reduce user interaction to a minimum. It fully uses the functionality of the mouse, menu and window system of the Symbolics Lisp Machine. When due to the incompleteness of the linguistic knowledge new information cannot be computed with full certainty, the system nevertheless goes ahead, using heuristics to present an 'educated guess' and notifying the user of this. These heuristics are based on linguistic as well as probabilistic data. A user monitoring the output of the constructor only needs to click on incorrect items or parts of items in the output (which is mouse-sensitive). This activates diagnostic procedures associated with the relevant linguistic objects. These procedures can delete erroneous objects already created, recompute them or transfer control to other objects. If the system can diagnose its error, a correction is presented. Otherwise, a menu of possible corrections (again constrained by heuristics) is presented from which the user may choose, or in the worst case (which is exceptional), the user has to enter the correct information himself.

An example of such a heuristic is the syllable boundary hypothesis generator of CHYP (section 5.2), which produces a best guess when not sure, while at the same

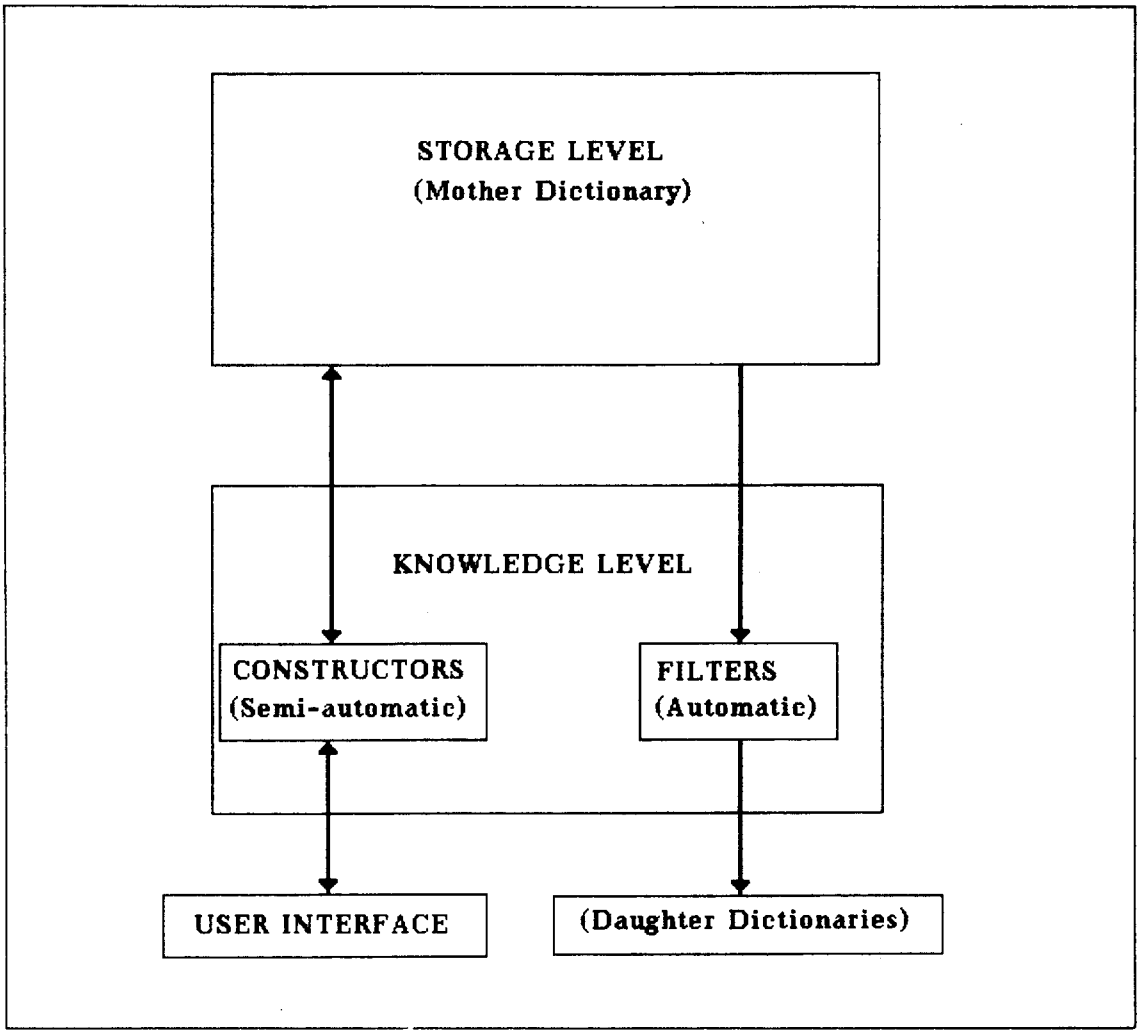


Figure 2. Architecture of the Flexible Dictionary System.

time keeping other possibilities available in decreasing order of probability. Another example, discussed in section 3.1, presents a best guess in the computation of vowel changes in irregular verb forms and in the interpretation of grapheme <e> as /∂/, /e/ or /ϵ/, while again keeping track of alternative possibilities.

8.2.3 Construction of a Rhyme Dictionary

Automatic *dictionary construction* can be easily done by using a particular filter (e.g., a citation form dictionary can be filtered out from a word form dictionary). Other more complex constructions can be achieved by combining a particular constructor or set of constructors with a filter. For example, to generate a word form lexicon on the basis of a citation form lexicon, we first have to apply a constructor to it (morpho-

logical synthesis), and afterwards filter the result into a suitable form. In this section, we will describe how a *rhyme dictionary* can be constructed on the basis of a spelling word form lexicon in an attempt to point out how the FDS can be applied advantageously in lexicography.

First, a *constructor* must be defined for the computation of a broad phonetic transcription of the spelling forms. This involves primitive linguistic procedures for syllabification (section 4.1), phonemisation and stress assignment or stress retrieval (section 4.2). The phonemisation algorithm should be adapted in this case by removing a number of irrelevant phonological rules (e.g., assimilation rules). The result of applying this constructor is the extension of each entry in the source dictionary with an additional field (or slot at the knowledge level) for the transcription. Next, a filter object is defined working in three steps:

(i) Take the broad phonetic transcription of each dictionary entry and reverse it.

(ii) Sort the reversed transcriptions first according to their rhyme determining part and then alphabetically. The rhyme determining part consists of the nucleus and coda of the last stressed syllable and the following weak syllables if any. For example, the rhyme determining part of *wérvelen* (to whirl) is *er-ve-len*, of *versnéllen* (to accelerate) *el-len*, and of *óverwérk* (overwork) *erk*.

(iii) Print the spelling associated with each transcription in the output file. The result is a spelling rhyme dictionary. If desirable, the spelling forms can be accompanied by their phonological transcription.

Using the same information, we can easily develop an alternative filter which takes into account the metre of the words as well. Although two words rhyme even when their rhythm (defined as the succession of stressed and unstressed syllables) is different, it is common poetic practice to look for rhyme words with the same metre. The metre frame can be derived from the phonetic transcription. In this variant, step (ii) must be preceded by a step in which the (reversed) phonetic transcriptions are sorted according to their metre frame.

8.2.4 Related Research

The presence of both static information (morphemes and features) and dynamic information (morphological rules) in LDBs is also advocated by Domenig and Shann (1986). Their prototype includes a morphological 'shell' making possible real time word analysis when only stems are stored. This morphological knowledge is not

used, however, to extend the dictionary and their system is committed to a particular formalism while ours is notation-neutral and unrestrictedly extendible due to the object-oriented implementation.

The LDB model outlined in Isoda, Aiso, Kamibayashi and Matsunaga (1986) shows some similarity to our filter concept. Virtual dictionaries can be created using base dictionaries (physically existing dictionaries) and user-defined Association Interpreters (AIPs). The latter are programs which combine primitive procedures (pattern matching, parsing, string manipulation) to modify the fields of the base dictionary and transfer control to other dictionaries. This way, for example, a virtual English-Japanese synonym dictionary can be created from English-English and English-Japanese base dictionaries. In our own approach, all information available is present in the same MD, and filters are used to create base dictionaries (physical, not virtual). Constructors are absent in the architecture of Isoda et al. (1986).

Johnson (1985) describes a program computing a reconstructed form on the basis of surface forms in different languages by undoing regular sound changes. The program, which is part of a system compiling a comparative dictionary (semi-)automatically, may be interpreted as related to the concept of a constructor in our own system, with construction limited to simple string manipulations, and not extensible unlike our own system.

8.2.5 Conclusion

We see three main advantages in our approach. First, the theoretical distinction between a *dynamic linguistic level* with a practical and user-friendly interface and a *static storage level* allows us to construct, extend and maintain a large MD quickly, conveniently and cost-effective (at least for those linguistic data of which the rules are fairly well understood). Obviously, MDs of different languages will not contain the same information: while it may be feasible to incorporate inflected forms of nouns, verbs and adjectives in it for Dutch, this would not be the case for Finnish. Second, the linguistic knowledge necessary to build constructor objects can be tested, optimised and experimented with by continuously applying it to large amounts of lexical material. Third, optimal LDBs for specific applications (e.g. hyphenation, spelling error correction etc.) can be easily derived from the MD due to the introduction of *filters* which automatically derive DDs.

It may be the case that our approach cannot be easily extended to the domain of syntactic and semantic dictionary information. It is not obvious how constructors could be built, for instance, for the (semi-)automatic computation of case frames for verbs or of semantic representations for compounds. Yet, our heuristics-driven cooperative interface could be profitably used in these areas as well.

8.3 More Applications

In the remainder of this chapter, we will indicate some additional practical systems in which our morpho-phonological model or applications already described may feature.

The Lexical Analyser. A lexical analyser is a pre-processor for natural language interpreting systems (parsers, retrieval systems, text-to-speech systems etc.). It transforms 'raw text' containing special symbols, punctuation, dates, numbers, names, addresses etc. into a representation which is convenient for further processing.

To achieve this effect, the lexical analyser should incorporate a spelling and typing error detection module (Chapter 6), and several *special-purpose parsers* whose task it is, on the one hand, to filter out potentially dangerous text features (e.g. idioms and spelling errors) and on the other hand to provide early semantic information to help the 'higher' interpretation processes in difficult cases.

Special purpose parsers should be defined for at least the following text features:

(i) *Dates*: Dates in different formats (*January the third, Third of January, 3/1*) should be transformed into the same format, intelligible to the semantic interpretation component.

(ii) *Names*: Names provide a lot of information for semantic interpretation (mostly, they are linked to a concept to which information is attached). In some cases, names can be detected early.

(iii) *Formatting codes*: These codes are special symbols which are often concatenated to the words (e.g. for bolding or italicising). A parser cannot recognise words if these codes are still added to them. We therefore need a special purpose parser to remove them.

(iv) *Punctuation*: Punctuation can be concatenated to words as well. It is easy enough to strip punctuation marks from words and forget about them. A better solution, however, would be to keep them in the internal representation for use by the

syntactic parser.

Recent versions of our spelling checker incorporate (iii) and (iv). By way of example, an input sentence (1) would obtain an internal representation like (2) before being passed on to the syntactic analysis.

(1) Ever since 4/4/1983 Jones had been missign, but the third of June he turned up again.

(2) (:capital ever since :date 04041983 :capital :name jones had been :error missign :comma but :date 03061983 he turned up again :fullstop)

The importance of a lexical analyser should not be underestimated. Any practical system which takes text as input should incorporate one, and the development of some special-purpose parsers is far from trivial.

Text-to-Speech Systems. The phonemisation system (section 4.2) can be used as a module in a general text-to-speech (speech synthesis) system. Such a system consists of a number of stages (cp. Hill, 1980; Allen, 1980; Simon, 1980). The linguistic part (i and ii) is the hardest if high-quality speech from unrestricted text is needed. The phonetic/technological part (iii and iv) is much better understood. Figure 3 illustrates the interrelations between the different linguistic sub-modules. The shaded boxes represent modules which we have implemented and described in earlier chapters.

(i) *Text pre-processing*: numbers and abbreviations are expanded, spelling checked, and exception dictionaries consulted (cf. the description of the *lexical analyser* earlier in this chapter).

(ii) *Phonemisation*: a system like ours, augmented with information about syntactic structure, would be able to achieve high-quality transliteration. As was pointed out in section 4.2, phonemisation requires the activation of the morphological analyser (detection of internal word boundaries) and the syllabification module (syllable structure is necessary if not essential in phonemisation). Syntactic structure (not implemented in our system) is necessary to constrain some phonological processes and to compute intonation contours.

(iii) *Parametric representation*: individual phonemes must be transformed into sets of parameters. Typical parameters include voicing frequency, voicing amplitude, fricative amplitude, aspiration amplitude, formant frequency and main frequency peak for fricative energy.

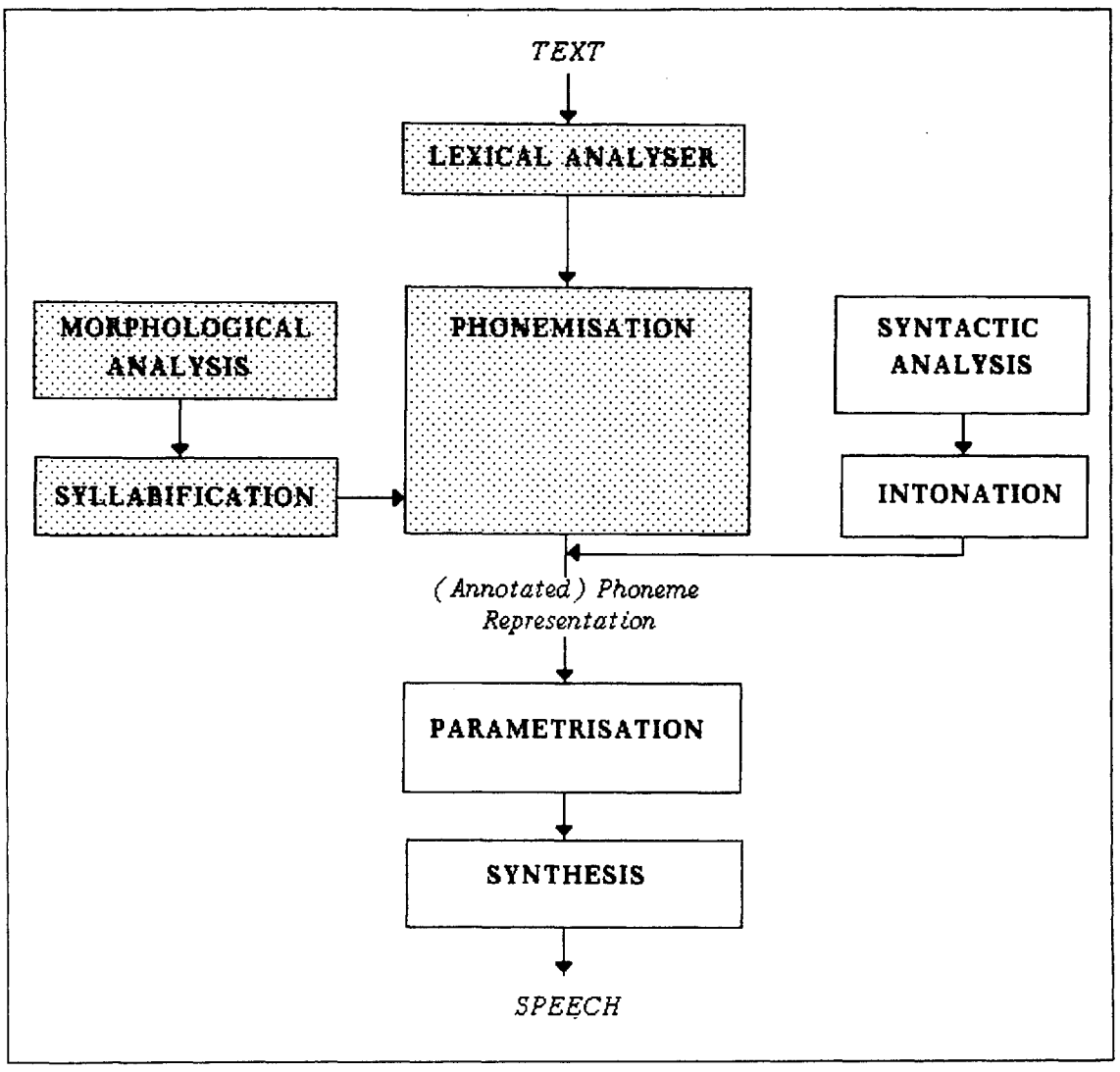


Figure 3. Modules in a text-to-speech system.

(iv) *Waveform construction*: by means of a synthesiser, waveforms must be generated on the basis of the parametric phoneme representations and additional intonation and stress information.

Applications for a text-to-speech system abound: they are a crucial component in the production part of natural language interfaces (e.g. to data-bases and computer systems). Speech interfaces make the computer accessible to handicapped and (computer-)illiterate persons. Other obvious applications (in descending order of importance) include reading machines for the blind, teaching machines, security warning systems, talking calculators, talking washing machines, talking cars etc.

8.4 Conclusion

In this chapter, the applicability of our morpho-phonological model in the construction of rule-testing systems, lexicographical environments, lexical analysers and text-to-speech systems was described. We could have added sections on the role of a model like ours in machine translation, dialogue systems, typing aids, quantitative linguistics etc., but we have restricted ourselves to applications we have experimented with.

GENERAL CONCLUSION

A relation of mutual influence exists between programming and writing about programs. Writing a text about a program inspires modifications to the program. When these modifications are implemented, several shortcomings in the text are brought to light. This calls for a new version of the text which inspires new modifications to the program and so on *ad infinitum*. This loop can only be ended by a deadline.

A central claim in this dissertation was that algorithms and heuristics for language technology should be based on a computer model which is motivated by (psycho-)linguistic research. Restricted linguistic knowledge is indispensable for the construction of even the most simple language technological applications. Furthermore, it was argued that text processing algorithms developed for one language cannot be simply transferred to other languages without essential modifications. We have gathered evidence for these theses in Chapters 5 and 6. We noted there that the spelling behaviour of Dutch compounds makes necessary a level of morphological analysis in both automatic hyphenation and automatic spelling and typing error detection and correction. Incidentally, it may be a good idea to return to a version of 17th century Dutch spelling, in which internal word boundaries were indicated with a hyphen (for example *Ont-werp der Neder-duitsche letter-konst* of Johannes Troost, 1649). Such a spelling change would make word processing software for Dutch a lot simpler.

Throughout the text, the position was implicitly defended that when weighing the different aspects which constitute the concept 'computationally efficient': speed, memory requirements and accuracy, most attention should be paid to accuracy, as hardware possibilities evolve at such a rate that it would be foolish to make theoretical decisions on the basis of temporary physical limitations. This made us look for principled solutions to language technological problems rather than concentrate on efficiency boosting heuristics.

We also stressed the importance of user-friendly interfaces to applications and of error recovery heuristics (this position was particularly defended in the sections on the rule testing device, the lexicographic environment and the Intelligent Tutoring System).

In the course of building an application layer around the morphophonological module, several prototypes of programs were built. We will finish by reviewing them here concisely.

(i) A (nameless) automatic syllabification program using morphological analysis divides words into syllables with nearly perfect accuracy (99.99% in a limited test, more experimentation is necessary). In principle, this approach allows the correct hyphenation of all existing and possible Dutch words. Remaining errors are due to limitations of the morphological parsing algorithm. A serious drawback of this program is the fact that it needs a complete word form dictionary in external memory.

(ii) *CHYP* (for cautious hyphenation program) was based on a statistical study of syllable structure and on phonotactic restrictions. The program is small and efficient. In one mode, its accuracy equals that of current hyphenation programs which often include large amount of patterns or dictionary items, and makes the same mistakes. In another mode, it fails to insert a syllable boundary when not absolutely sure (with nearly perfect accuracy: 99.88%). CHYP is able to provide a full hyphenation (all boundaries) in 15% of polysyllabic words, at least two boundaries in 22% of polysyllabic words, and at least one in 63% of polysyllabic words. When applied to text-formatting, the average gain (measured with a spacing index) by using CHYP was 14% as opposed to 28% with a full hyphenation. The use of CHYP in optimising (i) and as an interactive hyphenation system was also argued.

(iii) *DSPELL* is a word level spelling and typing error detection program using morphological analysis to reduce overkill (flagging correct words as an error). This reduction was 10%. Undershoot was not measured.

(iv) *TDTDT* is a prototype of an ITS for teaching verbal inflections. The system is able to provide exercises, to react cooperatively to errors by the learner and to perform an elementary diagnosis of errors.

(v) *GRAFON* is an interactive phonological rule-testing devise, designed to help the phonologist in developing and debugging complex rule systems.

(vi) The *Flexible Dictionary System* is an environment in which lexicographers can create, extend and maintain lexical databases.

APPENDICES

Numbers of appendices refer to the Chapter in the text to which they belong. For example, A.3.1 is the appendix to Chapter 3 section 1.

APPENDIX A.3

A.3.1 Morphological Synthesis

We present here an overview of the concept hierarchy we use for morphological synthesis. This overview is split up in four diagrams (Figures A.1 to A.4). Black boxes indicate that there are sub-types of this concept which are not shown. These boxes are opened up in other diagrams. Figure A.5 provides an impression of the user interface which can be used by the linguist working with the synthesis program to inspect the different concepts in the system. The top frame of the window shows part of the concept hierarchy. The bottom right frame allows the user to compute forms. In the example, the spelling of *laten* (to let) and the lexical representation of *werken* (to work) are computed. The bottom left frame shows the internal structure of the verb form *laten*. The different subjects (attached to *laten* or inherited) are listed with their fillers. The structure of one of these fillers (verb-form #4) is shown in the bottom middle frame.

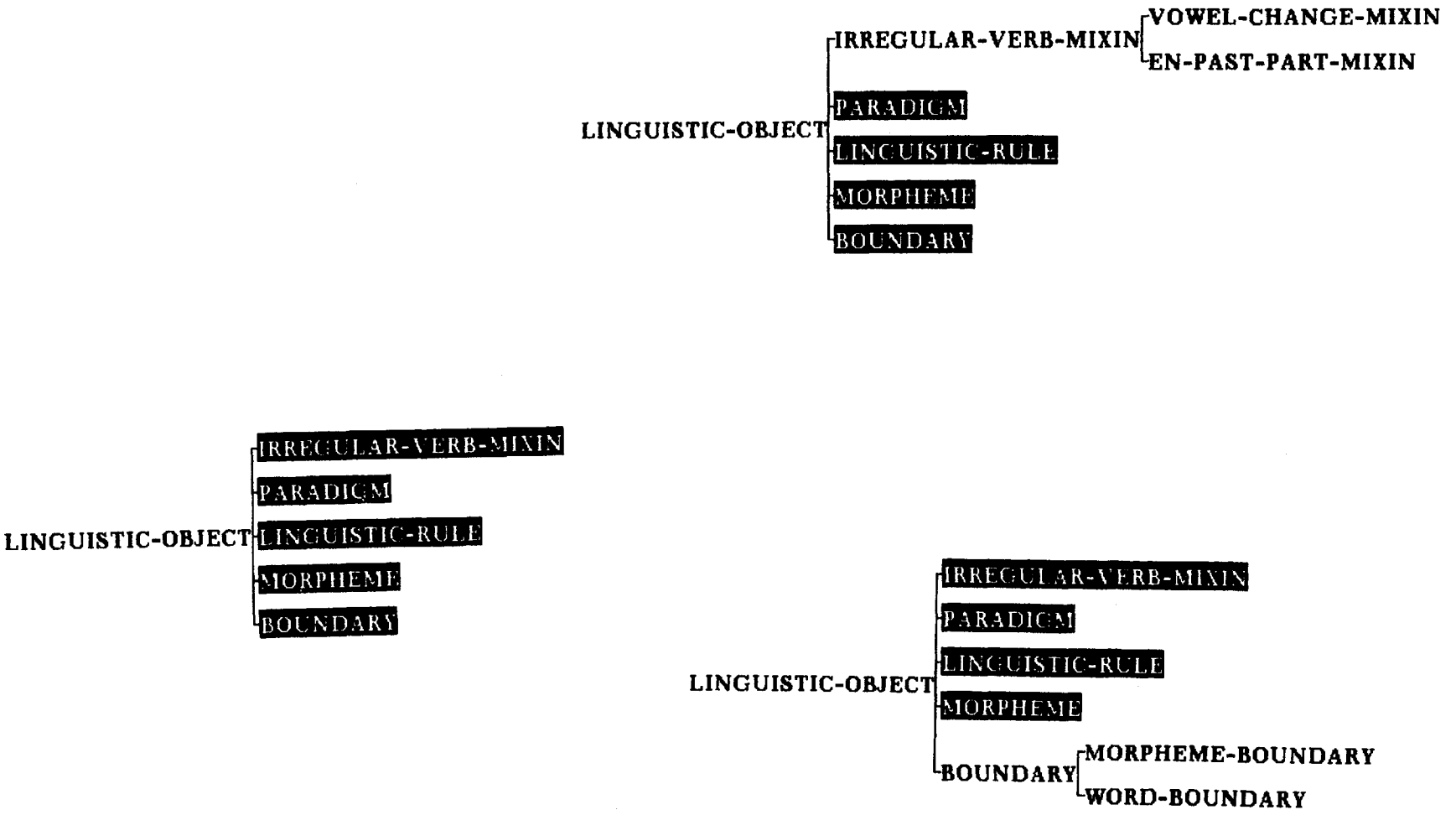
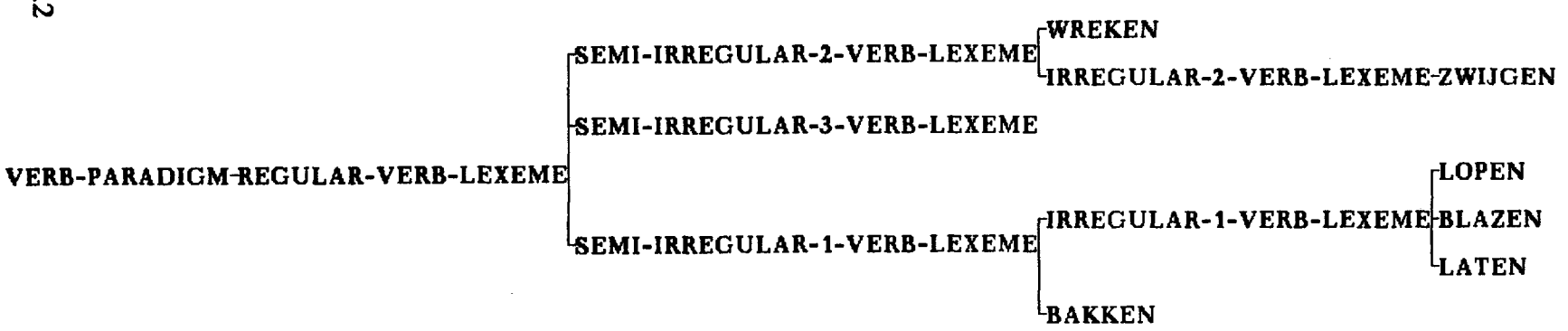

LINGUISTIC-OBJECT
IRREGULAR-VERB-MIXIN
VOWEL-CHANGE-MIXIN
EN-PAST-PART-MIXIN
PARADIGM
LINGUISTIC-RULE
MORPHEME
BOUNDARY
LINGUISTIC-OBJECT
IRREGULAR-VERB-MIXIN
PARADIGM
LINGUISTIC-RULE
MORPHEME
BOUNDARY
LINGUISTIC-OBJECT
IRREGULAR-VERB-MIXIN
PARADIGM
LINGUISTIC-RULE
MORPHEME
BOUNDARY
MORPHEME-BOUNDARY
WORD-BOUNDARY

Figure A.1

Figure A.2


VERB-PARADIGM
REGULAR-VERB-LEXEME
SEMI-IRREGULAR-2-VERB-LEXEME
WREKEN
IRREGULAR-2-VERB-LEXEME
ZWIJGEN
SEMI-IRREGULAR-3-VERB-LEXEME
SEMI-IRREGULAR-1-VERB-LEXEME
IRREGULAR-1-VERB-LEXEME
LOPEN
BLAZEN
LATEN
BAKKEN

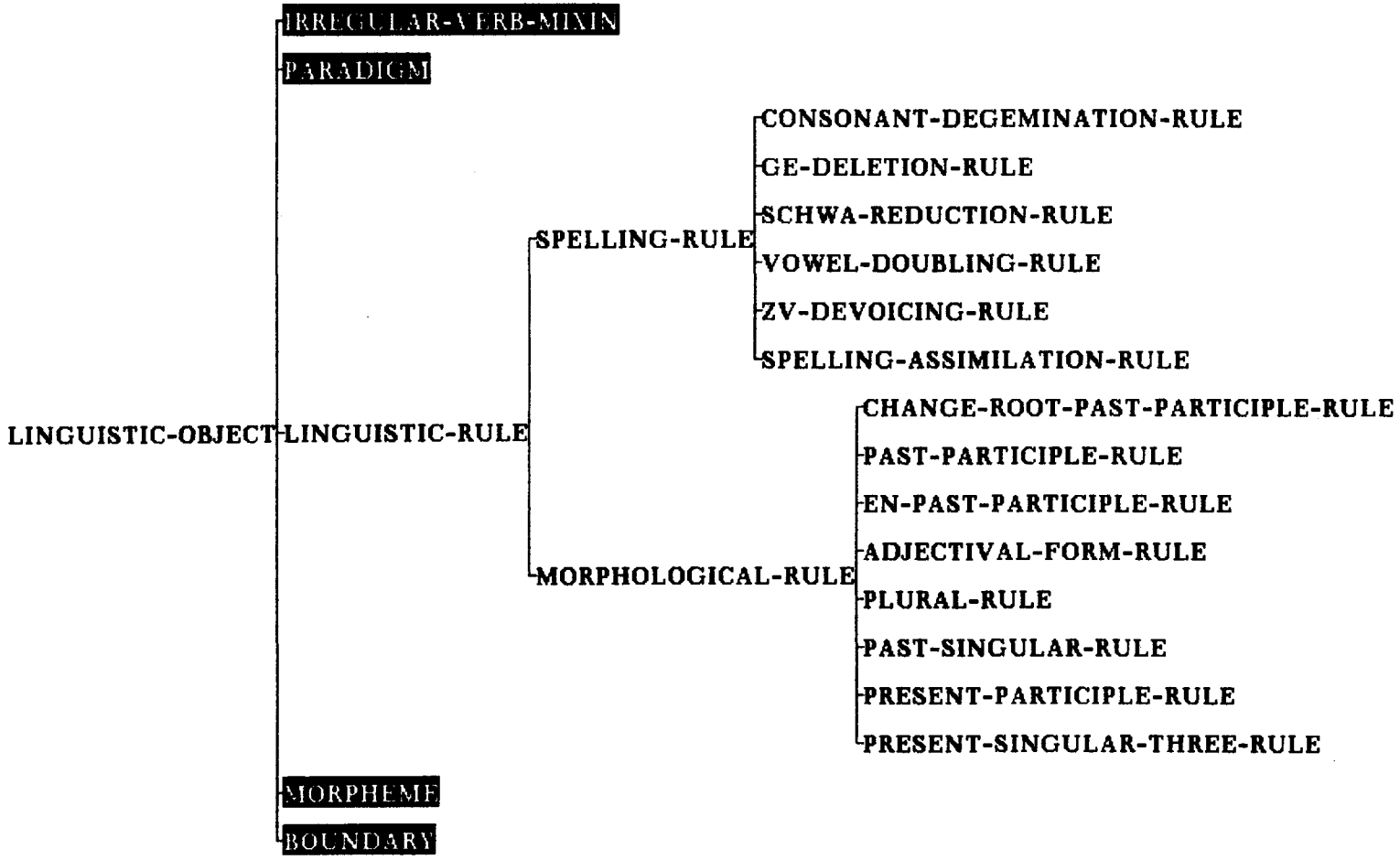

IRREGULAR-VERB-MIXIN
PARADIGM
CONSONANT-DEGEMINATION-RULE
GE-DELETION-RULE
SCHWA-REDUCTION-RULE
SPELLING-RULE
VOWEL-DOUBLING-RULE
ZV-DEVOICING-RULE
SPELLING-ASSIMILATION-RULE
CHANGE-ROOT-PAST-PARTICIPLE-RULE
PAST-PARTICIPLE-RULE
EN-PAST-PARTICIPLE-RULE
LINGUISTIC-OBJECT
LINGUISTIC-RULE
ADJECTIVAL-FORM-RULE
MORPHOLOGICAL-RULE
PLURAL-RULE
PAST-SINGULAR-RULE
PRESENT-PARTICIPLE-RULE
PRESENT-SINGULAR-THREE-RULE
MORPHEME
BOUNDARY

Figure A.3

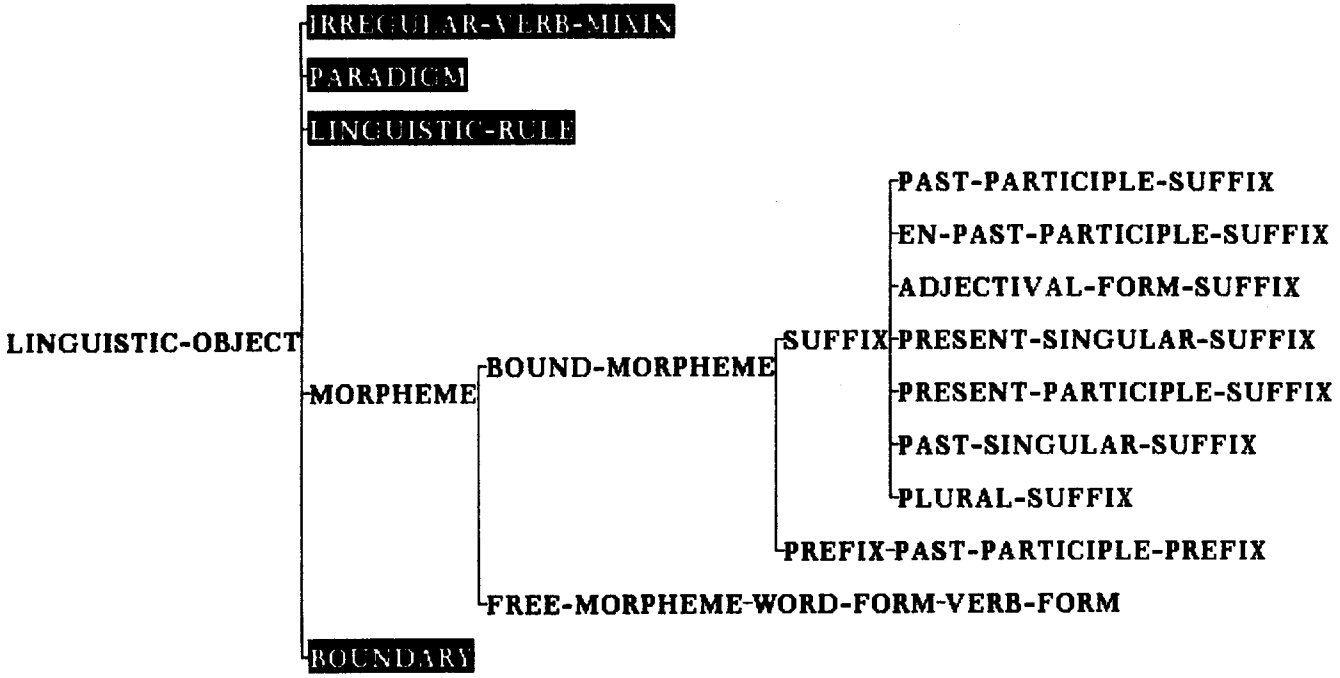

LINGUISTIC-OBJECT
IRREGULAR-VERB-MIXIN
PARADIGM
LINGUISTIC-RULE
MORPHEME
BOUND-MORPHEME
SUFFIX
PAST-PARTICIPLE-SUFFIX
EN-PAST-PARTICIPLE-SUFFIX
ADJECTIVAL-FORM-SUFFIX
PRESENT-SINGULAR-SUFFIX
PRESENT-PARTICIPLE-SUFFIX
PAST-SINGULAR-SUFFIX
PLURAL-SUFFIX
PREFIX
PAST-PARTICIPLE-PREFIX
FREE-MORPHEME
WORD-FORM-VERB-FORM
BOUNDARY

Figure A.4

Font
Krsi
Ctree
Cedit
Cinsp
Inter
Concep
Windo
Save-I
Load-I
Help
Concep
Messa

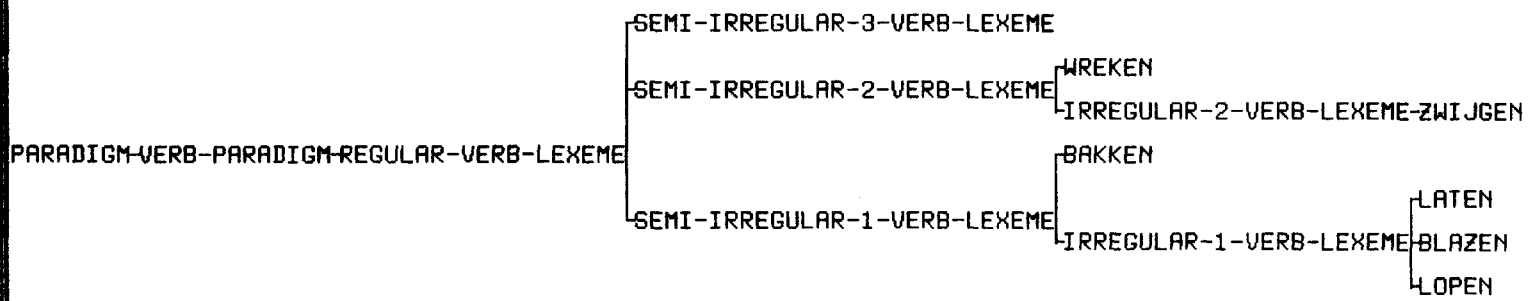


Figure A.5

tree3

Top of object

<LATEN>

META-CONCEPT-TYPE: <META-CONCEPT>
ROOT: <MORPHEME #1>
ADJECTIVAL-FORM: <VERB-FORM #2>
PAST-PARTICIPLE: <VERB-FORM #3>
PAST-PLURAL: <VERB-FORM #4>
PAST-SINGULAR: <VERB-FORM #5>
PRESENT-PARTICIPLE: <VERB-FORM #6>
PRESENT-PLURAL: <VERB-FORM #7>
PRESENT-SINGULAR-THREE: <VERB-FORM #8>
PRESENT-SINGULAR-TWO: <VERB-FORM #9>
PRESENT-SINGULAR-ONE: <VERB-FORM #10>
PARADIGM: <CONCEPT-LIST (<VERB-FOR
PAST-ROOT: <STRING liet>
CITATION-FORM: <STRING laten>
TYPE: <IRREGULAR-1-VERB-LEXEME

Bottom of object

Top of object

<VERB-FORM #4>

META-CONCEPT-TYPE: <META-CONCEPT>
SPELLING: <STRING lieten>
LEXICAL-REPRESENTATION: <STRING liet+
GRAMM-NUMBER: <PLURAL>
TENSE: <PAST>
FINITENESS: <FINITE>
LEXEME: <LATEN>
TYPE: <VERB-FORM>

Bottom of object

```
(>> referent spelling
     past-plural of laten)
"lieten"


(defconcept werken
  (a regular-verb-lexeme
     (citation-form [string "werken"])))
<CONCEPT-NAME #11>


(>> referent lexical-representation
     past-participle of werken)
"g@#werk+D"
[10:21:01 Hardcopy of Lisp Frame 1 has been
bitmapped to AGFA]
```

lisp1

krs system1

Next, we show some verb forms computed by our morphological synthesis system. For each form, the output of those spelling rules which modified the lexical representation are also given. Of each form, first the lexical representation is given, then the spelling rules applied to it (if any) and finally the spelling. Forms computed are present-singular-first, present-singular-third, present-singular-plural, present-participle, past-singular, past-plural and past-participle, in that order.

BAKKEN (to bake)

... bakk
... ... CONSONANT-DEGEMINATION-RULE: bak
... bak

... bakk+t
... ... CONSONANT-DEGEMINATION-RULE: bak+t
... bakt

... bakk+ə n
... bakken

... bakk+ə nd
... bakkend

... bakk#Də
... ... SPELLING-ASSIMILATION-RULE: bakk#tə
... ... CONSONANT-DEGEMINATION-RULE: bak#tə
... bakte

... bakk#Də+ə n
... ... SPELLING-ASSIMILATION-RULE: bakk#tə+ə n
... ... CONSONANT-DEGEMINATION-RULE: bak#tə+ə n
... ... SCHWA-REDUCTION-RULE: bak#tə+n
... bakten

... gə#bakk+ə n
... gebakken

HERSTELLEN (to repair)

... herstell
... ... CONSONANT-DEGEMINATION-RULE: herstel
... herstel

... herstell+t
... ... CONSONANT-DEGEMINATION-RULE: herstel+t

... herstelt

... herstell+∂n
... herstellen

... herstell+∂nd
... herstellend

... herstell#D∂
... ... SPELLING-ASSIMILATION-RULE: herstell#d∂
... ... CONSONANT-DEGEMINATION-RULE: herstel#d∂
... herstelde

... herstell#D∂+∂n
... ... SPELLING-ASSIMILATION-RULE: herstell#d∂+∂n
... ... CONSONANT-DEGEMINATION-RULE: herstel#d∂+∂n
... ... SCHWA-REDUCTION-RULE: herstel#d∂+n
... herstelden

... g∂#herstell+D
... ... SPELLING-ASSIMILATION-RULE: g∂#herstell+d
... ... CONSONANT-DEGEMINATION-RULE: g∂#herstel+d
... ... GE-DELETION-RULE: herstel+d
... hersteld

LATEN (to let)

... lat
... ... VOWEL-DOUBLING-RULE: laat
... laat

... lat+t
... ... VOWEL-DOUBLING-RULE: laat+t
... ... CONSONANT-DEGEMINATION-RULE: laa+t
... laat

... lat+∂n
... laten

... lat+∂nd
... latend

... liet
... liet

... liet+∂n
... lieten

... gə#lat+ən
... gelaten

POTTEN (to pot)

... pott
... ... CONSONANT-DEGEMINATION-RULE: pot
... pot

... pott+t
... ... CONSONANT-DEGEMINATION-RULE: pot+t
... ... CONSONANT-DEGEMINATION-RULE: po+t
... pot

... pott+ən
... potten

... pott+ənd
... pottend

... pott#Də
... ... SPELLING-ASSIMILATION-RULE: pott#tə
... ... CONSONANT-DEGEMINATION-RULE: pot#tə
... potte

... pott#Də+ən
... ... SPELLING-ASSIMILATION-RULE: pott#tə+ən
... ... CONSONANT-DEGEMINATION-RULE: pot#tə+ən
... ... SCHWA-REDUCTION-RULE: pot#tə+n
... potten

... gə#pott+D
... ... SPELLING-ASSIMILATION-RULE: gə#pott+
... ... CONSONANT-DEGEMINATION-RULE: gə#pot+
... gepot

VERVELEN (to bore)

... vervel
... ... VOWEL-DOUBLING-RULE: verveel
... verveel

... vervel+t
... ... VOWEL-DOUBLING-RULE: verveel+t
... verveelt

... vervel+ən

... vervelen

... vervel+ən
... vervelend

... vervel#Də
... ... SPELLING-ASSIMILATION-RULE: vervel#də
... ... VOWEL-DOUBLING-RULE: verveel#də
... verveelde

... vervel#Də+ən
... ... SPELLING-ASSIMILATION-RULE: vervel#də+ən
... ... VOWEL-DOUBLING-RULE: verveel#də+ən
... ... SCHWA-REDUCTION-RULE: verveel#də+n
... verveelden

The next form is wrong; *ver* has not been detected as a prefix, which results in the non-application of the ge-deletion-rule

... ge#vervel+D
... ... SPELLING-ASSIMILATION-RULE: gə#vervel+d
... ... VOWEL-DOUBLING-RULE: gə#verveel+d
... geverveeld

VERVELLEN (to peel)

... vervell
... ... CONSONANT-DEGEMINATION-RULE: vervel
... vervel

... vervell+t
... ... CONSONANT-DEGEMINATION-RULE: vervel+t
... vervelt

... vervell+ən
... vervellen

... vervell+ənd
... vervellend

... vervell#Də
... ... SPELLING-ASSIMILATION-RULE: vervell#də
... ... CONSONANT-DEGEMINATION-RULE: vervel#də
... vervelde

... vervell#Də+ən
... ... SPELLING-ASSIMILATION-RULE: vervell#də+ən

... ... CONSONANT-DEGEMINATION-RULE: vervel#də+ən
... ... SCHWA-REDUCTION-RULE: vervel#də+n
... vervelden

The remark made earlier about the computation of the past participle of *vervelen* applies here as well.

... gə#vervell+D
... ... SPELLING-ASSIMILATION-RULE: gə#vervell+d
... ... CONSONANT-DEGEMINATION-RULE: gə#vervel+d
... geverveld

A.3.2 Trace of Segmentation and Lookup

The following traces show the input and output for the functions *segment-string* and *lookup* for some input strings. Note that possible word forms which do not conform to the phonotactic restrictions of Dutch (e.g. *sdeursleutel* in the first example) are not looked up in the dictionary.

Output of (segment-string "huisdeursleutel");
SEGMENT-STRING and LOOKUP are traced.

```
1 Enter SEGMENT-STRING "huisdeursleutel"
| 1 Enter LOOKUP "huisdeursleutel"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "uisdeursleutel"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "isdeursleutel"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "deursleutel"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "eursleutel"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ursleutel"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "sleutel"
| 1 Exit LOOKUP (NOUN)
| 2 Enter SEGMENT-STRING "huisdeur"
|   1 Enter LOOKUP "huisdeur"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "uisdeur"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "isdeur"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "deur"
|   1 Exit LOOKUP (NOUN)
|   3 Enter SEGMENT-STRING "huis"
|   | 1 Enter LOOKUP "huis"
|   | 1 Exit LOOKUP (NOUN)
|   3 Exit SEGMENT-STRING ("huis")
| 2 Exit SEGMENT-STRING ("huis" "deur")
1 Exit SEGMENT-STRING ("huis" "deur" "sleutel")
```

Output of (segment-string "vermogensbelasting");
SEGMENT-STRING and LOOKUP are traced.

```
1 Enter SEGMENT-STRING "vermogensbelasting"
| 1 Enter LOOKUP "vermogensbelasting"
```

```
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ermogensbelasting"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "mogensbelasting"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ogensbelasting"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "gensbelasting"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ensbelasting"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "belasting"
| 1 Exit LOOKUP (NOUN)
| 2 Enter SEGMENT-STRING "vermogens"
|   1 Enter LOOKUP "vermogens"
|   1 Exit LOOKUP (VERB NOUN)
| 2 Exit SEGMENT-STRING ("vermogens")
1 Exit SEGMENT-STRING ("vermogens" "belasting")
```

Output of (segment-string "huisvuilvernietigingsfabriek");
SEGMENT-STRING and LOOKUP are traced.

```
1 Enter SEGMENT-STRING "huisvuilvernietigingsfabriek"
| 1 Enter LOOKUP "huisvuilvernietigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "uisvuilvernietigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "isvuilvernietigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "vuilvernietigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "uilvernietigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ilvernietigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "vernietigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ernietigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "nietigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ietigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "etigingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "tigingsfabriek"
| 1 Exit LOOKUP -
```

```
| 1 Enter LOOKUP "igingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "gingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ingsfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "sfabriek"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "fabriek"
| 1 Exit LOOKUP (NOUN)
| 2 Enter SEGMENT-STRING "huisvuilvernietigings"
|   1 Enter LOOKUP "huisvuilvernietigings"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "uisvuilvernietigings"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "isvuilvernietigings"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "vuilvernietigings"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "uilvernietigings"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "ilvernietigings"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "vernietigings"
|   1 Exit LOOKUP (NOUN)
|   3 Enter SEGMENT-STRING "huisvuil"
|   | 1 Enter LOOKUP "huisvuil" T
|   | 1 Exit LOOKUP (NOUN)
|   3 Exit SEGMENT-STRING ("huisvuil")
| 2 Exit SEGMENT-STRING ("huisvuil" "vernietigings")
1 Exit SEGMENT-STRING ("huisvuil" "vernietigings" "fabriek")
```

Output of (segment-string "boeredochtersavondgebed");
SEGMENT-STRING and LOOKUP are traced.

```
1 Enter SEGMENT-STRING "boeredochtersavondgebed"
| 1 Enter LOOKUP "boeredochtersavondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "oeredochtersavondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "eredochtersavondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "redochtersavondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "edochtersavondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "dochtersavondgebed"
```

```
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ochtersavondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "tersavondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ersavondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "savondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "avondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "vondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "ondgebed"
| 1 Exit LOOKUP -
| 1 Enter LOOKUP "gebed"
| 1 Exit LOOKUP (NOUN)
| 2 Enter SEGMENT-STRING "boeredochtersavond"
|   1 Enter LOOKUP "boeredochtersavond"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "oeredochtersavond"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "eredochtersavond"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "redochtersavond"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "edochtersavond"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "dochtersavond"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "ochtersavond"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "tersavond"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "ersavond"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "savond"
|   1 Exit LOOKUP -
|   1 Enter LOOKUP "avond"
|   1 Exit LOOKUP (NOUN)
|   3 Enter SEGMENT-STRING "boeredochters"
|   | 1 Enter LOOKUP "boeredochters"
|   | 1 Exit LOOKUP -
|   | 1 Enter LOOKUP "oeredochters"
|   | 1 Exit LOOKUP -
|   | 1 Enter LOOKUP "eredochters"
|   | 1 Exit LOOKUP -
```

```
|   | 1 Enter LOOKUP "redochters"
|   | 1 Exit LOOKUP -
|   | 1 Enter LOOKUP "edochters"
|   | 1 Exit LOOKUP -
|   | 1 Enter LOOKUP "dochters"
|   | 1 Exit LOOKUP (NOUN)
|   | 4 Enter SEGMENT-STRING "boere"
|   |   1 Enter LOOKUP "boere"
|   |   1 Exit LOOKUP (NOUN VERB)
|   | 4 Exit SEGMENT-STRING ("boere")
|   3 Exit SEGMENT-STRING ("boere" "dochters")
| 2 Exit SEGMENT-STRING ("boere" "dochters" "avond")
1 Exit SEGMENT-STRING ("boere" "dochters" "avond" "gebed")
```

APPENDIX A.4

A.4.1 Source Text

(By Kees Fens, from KUZIEN, Summer 1985)

Houd ik van mijn kamer? Twee glaswanden met luxaflex, een grijze muur van ruwe stenen en een witte wand van even ruwe stenen. Dat vertedert niet. Over het lattenplafond lopen drie rijen Tee-El-buizen, en alleen in de zwartste tijd van het jaar neemt het licht ook bezit van de kamer; meestal houdt het zich op koele afstand. Er staan in de kamer drie kasten van beige-grijs metaal; ze zijn altijd gesloten. Ik heb met hen in een kleine drie jaar geen relatie kunnen opbouwen. Er zijn een bureau en en vergadertafel, beide voortgebracht door de lege geest van een kantoormeubelfabrikant. Dan zijn er twee planten, introverten, met misschien fraaie groene gedachten, maar ze laten ze niet blijken. Het waait nooit in de kamer, de twee weten al lang niet meer dat ze tot de natuur horen. Aan de witte wand hangen twee schilderijen: Holbeins getekende portret van Thomas More; More kijkt peinzend voor zich uit, soms denk ik met hem mee, en dan belanden we in Charterhouse, want daar is hij het gelukkigst geweest, tot hij in de cel van de Tower kwam. Daarnaast hangt een zeer kleurrijk abstract schilderij van een goede vriend van mij; het is zo uitbundig dat het veel plezier in zichzelf moet hebben, zoals die vriend dat trouwens ook altijd heeft. Aan de grijze muur hangt een vooroorlogse tekening van een straatje in Amsterdam-oud-west. Door dat straatje ging ik zes jaar lang naar de lagere school, dagelijks zeer gelukkig: het was de mooiste tijd van mijn leven, omdat het leren tot niets diende. Op de voorgrond loopt een bakkertje achter een kar, ik heb hem gekend. Hij is in een plas graangestookte jenever verdronken. Op het bureau ligt bijna, op de tafel in het geheel niets. Ik houd niet van papier. Aan het opbouwen van een rijk gevoelsleven werkt de kamer niet mee. Hij is een onpersoonlijke ruimte, koud zelfs, zoals een publieke ruimte past. Een privé-kamer

vereist afsluiting. Maar sloten zijn in dit gebouw verboden. Ik houd niet van mijn kamer. Maar ik heb er drie weken van gehouden. Op donderdag achttien april, 's middags om drie uur kwamen er twee jongemannen mijn kamer binnen met een zin die ik veertig jaar geleden voor het laatst had gehoord: 'Dit gebouw is bezet. U dient deze kamer binnen een half uur te verlaten.' Ik begreep de onweerstaanbare kracht van de democratie en een uur later stond ik buiten. De al zo lege kamer liet ik leger achter. Vanaf de volgende dag stond de kamer drie weken in mijn hoofd. Ik zat aan de andere kant van het land, wat mijn betrokkenheid misschien groter maakte. Ik zag voortdurend de kamer, leeg, achtergelaten op een doodstille gang. En ik maakte er, in de geest, steeds een reis door heen. De grijze muur, de witte muur, zouden ze elkaar kunnen onderhouden? Op de negende dag zag ik op de witte muur een beurse plek, ontstaan door de tegen de muur slaande deur. Steen kan vertederen. Het licht was uit, natuurlijk, zou More nog iets zien? Met de lampen aan kan hij in het vrolijke schilderij kijken, hij ziet nu slechts grijsheid. En zijn peinzen is overgegaan in een lichte melancholie. De bakker duwt zijn karretje het duister in en niet langer in de richting van de glorieuze brug waarachter mijn school lag. En de planten? Ik kreeg medelijden met hen. De een is een wat onnozele, hij lijkt ook niet te groeien, hij zal van de leegte niets begrijpen, maar de andere, die een boom probeert te zijn, die moet lijden onder de stilte van de dood die elke bezetting meebrengt. Ik trachtte de kamer te bewonen. Ik ging achter het lege bureau zitten en keek naar de kasten. Ze kunnen zich bezig houden met de boeken die erin staan, ik gaf ze toe: woordenboeken en naslagwerken zijn moeilijk te lezen bij slecht licht en opwindend zijn ze ook niet. Een kon zich vermaken met jaargangen van vele tijdschriften. Maar de derde. Die had niet veel anders dan aanzetten van scripties en daar kom je lege weken niet mee door. 'De telefoon slaapt op de lessenaar' dacht ik op de twaalfde dag, want poëzie kan misschien een leegte wat vullen. Hoe breng ik een teken van leven over naar ginds, naar de planten, de schilderijen, de kasten en de tafels? Ik heb twee keer het nummer van mijn kamer gebeld, laat in de avond, wanneer de verlatenheid het grootst moet zijn geweest. Ik hoopte dat de deur zou openstaan en de bel door de hele gang te horen zou zijn. Een collega mocht op de vijftiende dag van de bezetters iets van zijn kamer halen, onder geleide. Hij bewoont de kamer naast de mijne. De volgende dag belde hij mij: je kamer staat er bij zoals je hem hebt verlaten. Ik durfde niet te vragen naar de toestand van de meubels, de planten, More en het straatje (het kleurrijke schilderij gaf mij geen zorgen: die heeft genoeg aan zijn eigen geluk), want ik droeg mijn leed in stilte. Na drie weken kende ik van de kamer elke centimeter en elke vlek in de

vloerbekleding en op het tafelblad. Op maandag dertien mei kwam er een einde aan de bezetting. Om half twaalf in de morgen kwam ik de kamer binnen. Alles had zich weer op zichzelf teruggetrokken, de leegte was weer tot zijn oorspronkelijk gedaante gevuld. In de hoek achter een kast vond ik een paraplu. Die had ik helemaal vergeten. Met hem had ik nu het grootste medelijden. Ik ben mijn schuldgevoelens tegenover hem nog niet kwijt. Een voormalige bezetter komt vragen naar de datum van een college. Dat hij de kamer die hij zo deed lijden, zo maar in durfde, heeft mij het meest verbijsterd.

Kees Fens

A.4.2 Syllabification and Stress Assignment

##*houd##*ik##*van##*mijn##*ka=mer##
##*twee##*glas#*wan=den##*met##*lu=xaf=lex##
##en##*grij=ze##*muur##*van##*ru=we##*ste=nen##*en##
##en##*wit=te##*wand##*van##*e=ven##*ru=we##*ste=nen##
##*dat##ver#*te=dert##*niet##
##*o=ver##et##*lat=ten#*pla=fond##
##*lo=pen##*drie##*rij=en##*tee##*el##*bui=zen##
##*en##al=*leen##*in##de##*zwart=ste##*tijd##*van##et##*jaar##
##*neemt##et##*licht##ook##be#*zit##*van##de##*ka=mer##
##*mee=stal##*houdt##et##*zich##*op##*koe=le##*af#stand##
##er##*staan##*in##de##*ka=mer##*drie##*kas=ten##
##*van##*bei=ge##*grijs##me=*taal##
##ze##*zijn##*al=tijd##ge#*slo=ten##
##*ik##*heb##*met##*hen##*in##en##*klei=ne##*drie##*jaar##
##*geen##re=*la=tie##*kun=nen##*op#*bou=wen##
##er##*zijn##en##bu=*reau##*en##en##ver#*ga=der#*ta=fel##
##*bei=de##*voort#ge=*bracht##*door##de##*le=ge##*geest##
##*van##en##kan=*toor#*meu=bel#fa=bri=*kant##
##*dan##*zijn##er##*twee##*plan=ten##
##in=tro=*ver=ten##
##*met##mis=*schien##*fraai=e##*groe=ne##ge#*dach=ten##
##*maar##ze##*la=ten##ze##*niet##*blij=ken##
##et##*waait##*nooit##*in##de##*ka=mer##
##de##*twee##*we=ten##*al##*lang##*niet##*meer##
##*dat##ze##*tot##de##na=*tuur##*ho=ren##
##*aan##de##*wit=te##*wand##*han=gen##*twee##schil=de=*rij=en##
##*Hol=beins##ge#*te=ken=de##por=*tret##*van##*tho=mas##*mo=re##
##*mo=re##*kijkt##*pein=zend##*voor##*zich##*uit##
##*soms##*denk##*ik##*met##*hem##*mee##
##*en##*dan##be#*lan=den##we##*in##*char=ter=hou=se##

##*want##*daar##*is##*hij##et##ge#*luk=kigst##ge#*weest##
##*tot##*hij##*in##de##*cel##*van##de##*to=wer##*kwam##
##*daar#*naast##*hangt##en##*zeer##*kleur#*rijk##
ab=*stract##schil=de=*rij##
##*van##en##*goe=de##*vriend##*van##*mij##
##et##*is##*zo##uit#*bun=dig##
##*dat##et##*veel##ple=*zier##*in##*zich#*zelf##*moet##*heb=ben##
##*zo=*als##*die##*vriend##*dat##*trou=wens##
*ook##*al=tijd##*heeft##
##*aan##de##*grij=ze##*muur##*hangt##en##
*voo=roor=log=se##*te=ke=ning##
##*van##en##*straat#je##*in##*am=ster=dam##*oud##*west##
##*door##*dat##*straat#je##*ging##*ik##*zes##*jaar##*lang##
##*naar##de##*la=ge=re##*school##
##*da=ge=lijks##*zeer##ge=*luk=kig##
##et##*was##de##*moois=te##*tijd##*van##*mijn##*le=ven##
##om=*dat##et##*le=ren##*tot##*niets##*dien=de##
##*op##de##*voor=*grond##*loopt##en##*bak=ker#tje##
##*ach=ter##en##*kar##
##*ik##*heb##*hem##ge=*kend##
##*hij##*is##*in##en##*plas##*graan#ge#*stook=te##je=*ne=ver##
##ver#*dron=ken##
##*op##et##bu=*reau##*ligt##*bij#na##
##*op##de##*ta=fel##*in##et##ge=*heel##*niets##
##*ik##*houd##*niet##*van##pa=*pier##
##*aan##et##op#*bou=wen##*van##en##*rijk##ge=*voels#*le=ven##
##*werkt##de##*ka=mer##*niet##*mee##
##*hij##*is##en##on#per=*soon=lij=ke##*ruim=te##
##*koud##*zelfs##
##*zo#*als##en##pu=*blie=ke##*ruim=te##*past##
##en##pri=*ve##*ka=mer##ver=*eist##*af=slui=ting##
##*maar##*slo=ten##*zijn##*in##*dit##ge=*bouw##ver=*bo=den##
##*ik##*houd##*niet##*van##*mijn##*ka=mer##
##*maar##*ik##*heb##er##*drie##*we=ken##van##ge=*hou=den##
##*op##*don=der=dag##*acht#*tien##a=*pril##
##*smid=dags##*om##*drie##*uur##
##*kwa=men##er##*twee##*jon=ge#*man=nen##*mijn##*ka=mer##*bin=nen##
##*met##en##*zin##*die##*ik##*veer=tig##*jaar##ge=*le=den##
##*voor##et##*laatst##*had##ge=*hoord##
##*dit##ge=*bouw##*is##be=*zet##
##*u##*dient##*de=ze##*ka=mer##*bin=nen##
en##*half##*uur##te##ver=*la=ten##
##*ik##be=*greep##de##on=weer#*staan=ba=re##*kracht##
##*van##de##*de=mo=cra=tie##
##*en##en##*uur##*la=ter##
##*stond##*ik##*bui=ten##
##de##*al##*zo##*le=ge##*ka=mer##

##*liet##*ik##*le=ger##*ach=ter##
##*van=*af##de##*vol=gen=de##*dag##*stond##de##*ka=mer##
##*drie##*we=ken##*in##*mijn##*hoofd##
##*ik##*zat##*aan##de##*an=de=re##*kant##*van##et##*land##
##*wat##*mijn##be=*trok=ken=heid##mis=*schien##
##*gro=ter##*maak=te##
##*ik##*zag##*voort#*du=rend##de##*ka=mer##
##*leeg##
##*ach=ter#ge=*la=ten##*op##en##*dood#*stil=le##*gang##
##*en##*ik##*maak=te##er##
##*in##de##*geest##
##*steeds##en##*reis##*door##*heen##
##de##*grij=ze##*muur##
##de##*wit=te##*muur##
##*zou=den##ze##*el=kaar##*kun=nen##*on=der#*hou=den##
##*op##de##*ne=gen=de##*dag##
##*zag##*ik##*op##de##*wit=te##*muur##
##en##*beur=se##*plek##
##ont=*staan##*door##de##*te=gen##de##*muur##*slaan=de##*deur##
##*steen##*kan##ver=*te=de=ren##
##et##*licht##*was##*uit##
##na=*tuur=lijk##
##*zou##*mo=re##*nog##*iets##*zien##
##*met##de##*lam=pen##*aan##
##*kan##*hij##
##*in##et##*vro=lij=ke##schil=de=*rij##*kij=ken##
##*hij##*ziet##*nu##*slechts##*grijs=heid##
##*en##*zijn##*pein=zen##*is##*o=ver#ge=*gaan##
##*in##en##*lich=te##*me=lan=cho=lie##
##de##*bak=ker##*duwt##*zijn##*kar=re#tje##
##et##*duis=ter##*in##
##*en##*niet##*lan=ger##*in##de##*rich=ting##
##*van##de##*glo=rieu=ze##*brug##
##*waa=rach=ter##*mijn##*school##*lag##
##*en##de##*plan=ten##
##*ik##*kreeg##*me=de#*lij=den##*met##*hen##
##de##en##*is##en##*wat##on=*no=ze=le##
##*hij##*lijkt##*ook##*niet##te##*groei=en##
##*hij##*zal##*van##de##*leeg=te##*niets##be=*grij=pen##
##*maar##de##*an=de=re##
##*die##en##*boom##pro=*beert##te##*zijn##
##*die##*moet##*lij=den##
##*on=der##de##*stil=te##*van##de##*dood##
##*die##*el=ke##be=*zet=ting##*mee#*brengt##
##*ik##*tracht=te##de##*ka=mer##te##be=*wo=nen##
##*ik##*ging##*ach=ter##et##*le=ge##bu=*reau##*zit=ten##
##*en##*keek##*naar##de##*kas=ten##

##ze###*kun=nen###*zich###*be=zig#*hou=den##
##*met##de###*boe=ken##
##*die###*er#*in###*staan##
##*ik###*gaf##ze###*toe##
##*woor=den#*boe=ken###*en##na#*slag#*wer=ken##
##*zijn###*moei=lijk##te###*le=zen##
##*bij###*slecht###*licht##
##*en##op#*win=dend###*zijn##ze###*ook###*niet##
##en###*kon###*zich##ver#*ma=ken##
##*met###*jaar#*gan=gen##
##*van###*ve=le###*tijd#*schrif=ten##
##*maar##de###*der=de##
##*die###*had###*niet###*veel###*an=ders##
##*dan###*aan#*zet=ten###*van###*scrip=ties##
##*en###*daar###*kom##je###*le=ge###*we=ken###*niet###*mee###*door##
##de###*te=le=foon###*slaapt###*op##de###*les=se=naar##
##*dacht###*ik###*op##de###*twaalf=de###*dag##
##*want##po=e=*zie###*kan##mis=*schien##
en###*leeg=te###*wat###*vul=len##=
##*hoe###*breng###*ik##en###*te=ken###*van###*le=ven##
##*o=ver###*naar###*ginds##
##*naar##de###*plan=ten##
##de##schil=de=*rij=en##
##de###*kas=ten###*en##de###*ta=fels##
##*ik###*heb###*twee###*keer##
##et###*num=mer###*van###*mijn###*ka=mer##ge#*beld##
##*laat###*in##de###*a=vond##
##wan=*neer##de##ver=*la=ten=heid##
##et###*grootst###*moet###*zijn##ge=*weest##
##*ik###*hoop=te##
##*dat##de###*deur###*zou###*o=pen#*staan##
##*en##de###*bel###*door##de###*he=le##
*gang##te###*ho=ren###*zou###*zijn##
##en##col=*le=ga###*mocht###*op##de###*vijf#*tien=de###*dag##
##*van##de##be=*zet=ters##
##*iets##*van###*zijn###*ka=mer###*ha=len##
##*on=der##ge=*lei=de##
##*hij##be=*woont##de###*ka=mer###*naast##de###*mij=ne##
##de###*vol=gen=de###*dag###*bel=de###*hij###*mij##
##je###*ka=mer###*staat###*er#*bij##
##*zo#*als##je###*hem###*hebt##ver=*la=ten##
##*ik###*durf=de###*niet##te###*vra=gen##
##*naar##de###*toe#*stand###*van##de###*meu=bels##
##de###*plan=ten##
##*mo=re###*en##et###*straat#je##
##et###*kleur#*rij=ke##schil=de=*rij##
##*gaf###*mij###*geen###*zor=gen##

##*die###*heeft##ge=*noeg###*aan###*zijn##*ei=gen##ge=*luk##
##*want##*ik###*droeg##*mijn###*leed##*in##*stil=te##
##*na##*drie##*we=ken##
##*ken=de##*ik##*van##de##*ka=mer##
##*el=ke##*cen=ti#=*me=ter##
##*en##*el=ke##*vlek##*in##de##*vloer#be#*kle=ding##
##*en##*op##et##*ta=fel#*blad##
##*op##*maan=dag##*der=tien##*mei##
##*kwam##er##en##*ein=de##*aan##de##be#*zet=ting##
##*om##*half##*twaalf##
##*in##de##*mor=gen##
##*kwam##*ik##de##*ka=mer##*bin=nen##
##*al=les##*had##*zich##*weer##
##*op##*zich#*zelf##te=*rug#ge#*trok=ken##
##de##*leeg=te##*was##*weer##
##*tot##*zijn##oor=*spron=ke=lijk##ge=*daan=te##
##ge=*vuld##
##*in##de##*hoek##*ach=ter##en##*kast##
##*vond##*ik##en##pa=ra=*plu##
##*die##*had##*ik##*he=le=maal##ver=*ge=ten##
##*met##*hem##*had##*ik##*nu##
##et##*groot=ste##*me=de#*lij=den##
##*ik##*ben##*mijn##*schuld#ge=*voe=lens##
##*te=gen#*o=ver##*hem##
##*nog##*niet##*kwijt##
##en##*voor#*ma=li=ge##be=*zet=ter##
##*komt##*vra=gen##*naar##de##*da=tum##*van##en##col=*le=ge##
##*dat##*hij##de##*ka=mer##
##*die##*hij##*zo##*deed##*lij=den##
##*zo##*maar##*in##*durf=de##
##*heeft##*mij##et##*meest##ver=*bij=sterd##
##*kees##*fens##

A.4.3 Phonemisation

(Flemish variant)

'ɪk 'fɑ 'mɛⁱŋ 'kɑmər
'ɣlɑsˌwɑndə 'mɛt 'lʏksɑfləks əŋ 'ɣrɛⁱzə 'my:r 'vɑn 'rʏwə 'stenə 'ɛn
'ʌɪtə 'wɑnt 'fɑn 'evə 'rʏwə 'stenə
fər'tedərt 'nɪt
· ət 'lɑtəmˌplɑvɔnt 'lopə 'drɪ 'rɛⁱjə 'te 'ɛl 'bœʸzə 'ɛn ɑ'len 'ɪn də 'zwɑrstə
'fɑn ət 'jɑr 'nemt ət 'lɪχt og bə'zɪt 'fɑn də 'kɑmər
tɑl 'hɔᵘt ət 'sɪχ 'ɔp 'kulə 'ɑfstɑnt
tɑn 'ɪn də 'kɑmər 'drɪ 'kɑstə 'vɑm 'bɛⁱɣə 'ɣrɛⁱs mə'tɑl zə 'zɛⁱn 'ɑltɛⁱt χə'slotə
ɪɛp 'mɛt 'hɛn 'ɪn əŋ 'klɛⁱnə 'drɪ 'jɑr 'ɣen rə'lɑsɪ 'kœnə 'ɔˌbɔᵘwə
ɛⁱn əm by:'ro 'ɛn əɱ vər'ɣɑdərˌtɑfəl 'bɛⁱdə 'vortχəˌbrɑɣ 'dor də 'leɣə 'ɣest
əŋ kɑn'torˌməbəlfɑbriˌkɑnt
'zɛⁱn ər 'twe 'plɑntə ɪntro'vɛrtə 'mɛt mɪ'sχɪɱ 'frɑjə 'ɣrunə ɣə'dɑχtə
· zə 'lɑtə zə 'nɪd 'blɛⁱkə
vɑjt 'nojt 'ɪn də 'kɑmər də 'twe 'wetə 'ɑ 'lɑŋ 'nɪt 'mer 'dɑt sə 'tɔ də nɑ'ty:r
ə
ɖə 'wɪtə 'wɑnt 'hɑŋən 'twe sχɪldə'rɛⁱjə 'hɔlbɛⁱns χə'tekəndə pɔr'trɛt 'fɑn
ɑs 'morə
·ə 'kɛⁱkt 'pɛⁱnzənt 'for 'zɪχ 'œʸt 'sɔmz 'deŋk 'ɪk 'mɛt 'hɛ 'me
dɑm bə'lɑndə wə 'ɪŋ 'χɑrtərhɔᵘsə 'wɑn 'dɑr 'ɪs 'hɛⁱ ət χə'lækɪχst χə'west
'hɛⁱ 'ɪn də 'sɛl 'vɑn də 'towər 'kwɑm
ˌnɑst 'hɑŋt ən 'zer 'kløˌrɛⁱk ɑp'strɑkt sχɪldə'rɛⁱ 'vɑn əŋ 'ɣudə 'vrɪnt 'fɑ 'mɛⁱ
'so œʸd'bœndəχ 'dɑt ət 'fel plə'zi:r 'ɪn 'zɪχˌsɛləf 'mut 'hɛbə
vɑlz 'di 'vrɪn 'dɑ 'trɔᵘwəns 'ok 'ɑltɛⁱt 'heft
də 'ɣrɛⁱzə 'my:r 'hɑŋt əɱ 'vororlɔχsə 'tekənɪŋ 'vɑn ən 'strɑtjə 'ɪn 'ɑmstərdɑm
'wɛst
'dɑt 'strɑtjə 'ɣɪŋ 'ɪk 'sɛs 'jɑr 'lɑŋ 'nɑr də 'lɑɣərə 'sχol 'dɑɣələk 'ser ɣə'lækəχ
vɑz də 'mojstə 'tɛⁱt 'fɑ 'mɛⁱn 'levə ɔm'dɑt ət 'lerə 'tɔt 'nɪdz 'dɪndə
ɖə 'vorˌɣrɔnt 'lopt əm 'bɑkərcə 'ɑχtər əŋ 'kɑr 'ɪk 'hɛp 'hɛm ɣə'kent
'ɪs 'ɪn əm 'plɑs 'χrɑŋɣəˌstoktə jə'nevər vər'drɔŋkə
ɘd by:'ro 'lɪɣd 'bɛⁱnɑ 'ɔb də 'tɑfəl 'ɪn ət χə'hel 'nɪts
hɔᵘt 'nɪt 'fɑm pɑ'pi:r
ət ɔ'bɔᵘwə 'vɑn ən 'rɛⁱk χə'vulsˌlevə 'wɛrəg də 'kɑmər 'nɪt 'me
'ɪs ən ɔmpər'sonləkə 'rœʸmtə 'kɔᵘt 'sɛləfs 'zoˌwɑls əm py'blɪkə 'rœʸmtə 'pɑst
ɔri've 'kɑmər vər'ɛⁱst 'ɑfslœʸtɪŋ 'mɑr 'slotə 'zɛⁱn 'ɪn 'dɪt χə'bɔᵘw vər'bodə
hɔᵘt 'nɪt 'fɑ 'mɛⁱŋ 'kɑmər 'mɑr 'ɪk 'hɛp ər 'drɪ 'wekə vɑŋ ɣə'hɔᵘdə
dɔndərdɑχ 'ɑχˌtɪn ɑ'prɪl 'smɪdɑχs 'ɔm 'drɪ 'y:r
ɪmə ər 'twe 'jɔŋəˌmɑnə 'mɛⁱŋ 'kɑmər 'bɪnə
· ən 'zɪn 'di 'ɪk 'fertəχ 'jɑr ɣə'ledə 'vor ət 'lɑtst 'hɑt χə'hort

(Fundamental) fenszuid.out >walterd>old>grafon SYM1: (2) Font: A (PHONEME) * [More be]

t χə'bɔᵘw 'ɪz bə'zɛt
dɪn 'dezə 'kamər 'bɪnə ən 'haləf 'y:r tə vər'latə
bə'ɣreb də ɔŋwer'stambarə 'kraχt 'van də 'demokrasi 'ɛn ən 'y:r 'latər
ɔnt 'ɪg 'bœʸtə
'al 'zo 'leɣə 'kamər 'lit 'ɪk 'leɣər 'aχtər
n,av də 'vɔlɣəndə 'daχ 'stɔn də 'kamər 'dri 'wekə 'ɪ 'mɛⁱn 'hoft
'sat 'an də 'andərə 'kant 'fan ət 'lant 'wat 'mɛⁱm bə'trɔkənhɛⁱt mɪ'sχɪn
ɔtər 'maktə 'ɪk 'saχ 'for,dy:rən də 'kamər 'leχ 'aχtərɣə,latə 'ɔp ən 'dot,stɪlə 'ɣaŋ
'ɪk 'maktə ər 'ɪn də 'ɣest 'stets ən 'rɛⁱz 'dor 'hen
'ɣreⁱzə 'my:r də 'wɪtə 'my:r 'zɔᵘdə zə 'ɛlkar 'kœnə 'ɔndər,hɔᵘdə
də 'neɣəndə 'daχ 'zaχ 'ɪk 'ɔb də 'wɪtə 'my:r əm 'børsə 'plɛk
stan 'dor də 'teɣə də 'my:r 'slandə 'dør 'steŋ 'kaŋ vər'tedərə
'lɪχt 'was 'œʸt na'ty:rlək
'morə 'nɔχ 'it 'sɪn 'mɛ də 'lampə 'an 'kan 'hɛⁱ 'ɪn ət 'froləkə sχɪldə'rɛⁱ 'kɛⁱkə
'zɪt 'ny 'slɛχs 'χreⁱshɛⁱt 'ɛn 'zɛⁱm 'pɛⁱnzə 'ɪs 'ovərɣə,ɣan 'ɪn ən 'lɪχtə 'melaŋɣoli
'bakər 'dœwt 'sɛⁱŋ 'karəcə ə 'dœʸstər 'ɪn 'ɛ 'nit 'laŋər 'ɪn də 'rɪχtɪŋ
n də 'ɣlorizə 'brœχ 'waraχtər 'mɛⁱn 'sχo 'laχ
də 'plantə 'ɪ 'kreχ 'medə,lɛⁱdə 'mɛt 'hɛn də ən 'ɪs əŋ 'wat ɔ'nozələ
'lɛⁱkt 'ɔk 'ni tə 'ɣrujən 'hɛⁱ 'zal 'van də 'leχtə 'nidz bə'ɣreⁱpə
ar də 'andərə 'di əm 'bom pro'ber tə 'zɛⁱn 'di 'mut 'lɛⁱdə
dər də 'stɪltə 'van də 'dot
'ɛlkə bə'zɛtɪŋ 'me,brɛŋt
'traχtə də 'kamər tə bə'wonə 'ɪk 'χɪŋ 'aχtər ət 'leɣə by:'ro
ɔə 'ɛŋ 'kek 'nar də 'kastə
'kœnə 'zɪɣ 'bezɪχ,hɔᵘdə 'mɛ də 'bukə 'di 'ɛr,ɪn 'stan 'ɪk 'χaf sə 'tu
ɔrdəm,bukə 'ɛ na'slaχ,wɛrkə 'zɛⁱ 'mujlək tə 'lezə 'bɛⁱ 'sleχt 'lɪχt
ɔp'wɪndənt 'sɛⁱn zə 'ɔk 'nit
'kɔn 'zɪχ fər'makə 'mɛt 'jar,ɣaŋən 'vaŋ 'velə 'tɛⁱt,sχrɪftə 'mar də 'dɛrdə
'hat 'nit 'fel 'andərs 'dan 'an,zɛtə 'van 'skrɪpsɪs
'dar 'kɔm jə 'leɣə 'wekə 'nit 'me 'dor
'teləfon 'slapt 'ɔb də 'lɛsənar 'daχt 'ɪk 'ɔb də 'twaləvdə 'daχ
ɪnt powə'zi 'ka mɪ'sχɪn ən 'leχtə 'wat 'fœlə
'brɛŋ 'ɪk ən 'tekə 'van 'levə 'ovər 'nar 'ɣɪns 'nar də 'plantə də sχɪldə'rɛⁱjə
'kastə 'ɛn də 'tafəls
'hɛp 'twe 'ker ət 'nœmər 'va 'mɛⁱŋ 'kamər ɣə'bɛlt 'lat 'ɪn də 'avɔnt
'ner də vər'latənhɛⁱt ət 'χrotst 'mut 'sɛⁱŋ ɣə'west
'hoptə 'da də 'dør 'zɔᵘ 'opən,stan 'ɛn də 'bɛl 'dor də 'helə 'ɣaŋ tə 'horə 'zɔᵘ 'zɛⁱn
kɔ'leɣa 'mɔχt 'ɔb də 'vɛⁱf,tɪndə 'daχ 'van də bə'zɛtərs 'ɪts 'fan 'zɛⁱŋ 'kamər 'halə

(Fundamental) fenszuid.out >walterd>old>grafon SYM1: (2) Font: A (PHONEME) * [More ab
3:00 Hardcopy of Zmacs Frame 1 has been bitmapped to AGFA]

ˈɔndər ɣəˈlɛɪdə ˈhɛɪ bəˈwon də ˈkamər ˈnaz də ˈmɛɪnə
də ˈvɔlɣəndə ˈdɑɣ ˈbɛldə ˈhɛɪ ˈmɛɪ jə ˈkamər ˈstat ˈɛrˌbɛɪ ˈ
zoˌwals jə ˈhɛm ˈhɛpt fərˈlatə
ˈɪg ˈdœrəvdə ˈniˑ tə ˈvraɣə ˈnar də ˈtuˌstant ˈfan də ˈmøbəls də ˈplantə
ˈmorə ˈɛn ət ˈstratjə
ət ˈkløˌrɛɪkə sχɪldəˈrɛɪ ˈɣɑf ˈmɛɪ ˈɣen ˈzɔrɣə ˈdiˑ ˈheft χəˈnuχ ˈan ˈzɛɪn ˈɛɪɣə ɣəˈlœk
ˈwant ˈɪg ˈdruχ ˈmɛɪn ˈlet ˈɪn ˈstɪltə
ˈna ˈdriˑ ˈwekə ˈkɛndə ˈɪk ˈfan də ˈkamər ˈɛlkə ˈsɛnsiˌmetər
ˈɛn ˈɛlkə ˈvlɛk ˈɪn də ˈvlɑːrbəˌkledɪŋ ˈɛn ˈɔp ə ˈtafəlˌblɑt
ˈɔp ˈmandɑɣ ˈdɛrtiˑ ˈmɛɪ ˈkwam ər ən ˈɛɪndə ˈan də bəˈzetɪŋ
ˈɔm ˈhaləf ˈtwaləf ˈɪn də ˈmɔrɣə ˈkwam ˈɪg də ˈkamər ˈbɪnə
ˈɑləs ˈhɑt ˈsɪχ ˈwer ˈɔp ˈsɪχˌsɛləf təˈrœχəˌtrɔkə də ˈleχtə ˈwɑs ˈwer
ˈtɔt ˈsɛɪn orˈsprɔŋkələk χəˈdantə ɣəˈvœlt
ˈɪn də ˈhuk ˈɑχtər əŋ ˈkɑst ˈvɔnt ˈɪk əm pɑrɑˈply ˈdiˑ ˈhɑt ˈɪk ˈheləmal vərˈɣetə
ˈmɛt ˈhɛm ˈhɑt ˈɪk ˈny ət ˈχrotstə ˈmedəˌlɛɪdə
ˈɪg ˈbɛ ˈmɛɪn ˈsχœltχəˌvuləns ˈteɣənˌovər ˈhɛm ˈnɔχ ˈnɪt ˈkwɛɪt
əŋ ˈvorˌmaliˑɣə bəˈzɛtər ˈkɔmt ˈfraɣə ˈnar də ˈdatœm ˈvan əŋ kɔˈleɣə
ˈdɑt ˈhɛɪ də ˈkamər ˈdiˑ ˈhɛɪ ˈzo ˈdet ˈlɛɪdə ˈzo ˈmar ˈɪn ˈdœrəvdə
ˈheft ˈmɛɪ ət ˈmest fərˈbɛɪstərt

ˈkes ˈfɛns

ACS (Fundamental) fenszuid.out >walterd>old>grafon SYM1: (2) Font: A (PHONEME) * [More ab
.:13:48 Hardcopy of Zmacs Frame 1 has been bitmapped to AGFA]

A.4.4 Phonemisation

(Dutch variant)

ˈhɔᵘt ˈɪk ˈfɑ ˈmɛⁱŋ ˈkɑmər
ˈtweⁱ ˈχlɑsˌwɑndə ˈmɛt ˈlʏksɑfləks əŋ ˈχrɛⁱsə ˈmyᵊr ˈfɑn ˈrywə ˈsteⁱnə ˈɛn
əm ˈwɪtə ˈwɑnt ˈfɑn ˈeⁱfə ˈrywə ˈsteⁱnə
ˈdɑt fərˈteⁱdərt ˈnɪt
ˈoᵘfər ət ˈlɑtəmˌplɑvɔnt ˈloᵘpə ˈdrɪ ˈrɛⁱjə ˈteⁱ ˈɛl ˈbœʸsə ˈɛn ɑˈleⁱn ˈɪn də ˈswɑrstə
ˈtɛⁱt ˈfɑn ət ˈjɑr ˈneⁱmt ət ˈlɪχt oᵘg bəˈsɪt ˈfɑn də ˈkɑmər
ˈmeⁱstɑl ˈhɔᵘt ət ˈsɪχ ˈɔp ˈkulə ˈɑfstɑnt
ər ˈstɑn ˈɪn də ˈkɑmər ˈdrɪ ˈkɑstə ˈfɑm ˈbɛⁱχə ˈχrɛⁱs məˈtɑl
sə ˈsɛⁱn ˈɑltɛⁱt χəˈsloᵘtə
ˈɪk ˈhɛp ˈmɛt ˈhɛn ˈɪn əŋ ˈklɛⁱnə ˈdrɪ ˈjɑr ˈχeⁱn rəˈlɑtsɪ ˈkœnə ˈɔˌbɔᵘwə
ər ˈsɛⁱn əm byᵊˈroᵘ ˈɛn əŋ fərˈχɑdərˌtɑfəl ˈbɛⁱdə ˈfoᵘrtχəˌbrɑɣ ˈdoᵘr də ˈleⁱχə ˈχeⁱst
ˈfɑn əŋ kɑnˈtoᵘrˌmøʸbəlfɑbriˌkɑnt
ˈdɑn ˈsɛⁱn ər ˈtweⁱ ˈplɑntə
ɪntroᵘˈfɛrtə
ˈmɛt mɪˈsχɪm ˈfrɑjə ˈχrunə χəˈdɑχtə ˈmɑr sə ˈlɑtə sə ˈnɪd ˈblɛⁱkə
ət ˈwɑjt ˈnoᵘjt ˈɪn də ˈkɑmər də ˈtweⁱ ˈweⁱtə ˈɑ ˈlɑŋ ˈnɪt ˈmeⁱr
ˈdɑt sə ˈtɔ də nɑˈtyᵊr ˈhoᵘrə
ˈɑn də ˈwɪtə ˈwɑnt ˈhɑŋən ˈtweⁱ sχɪldəˈrɛⁱjə
ˈhɔlbɛⁱns χəˈteⁱkəndə pɔrˈtrɛt ˈfɑn ˈtoᵘmɑs ˈmoᵘrə
ˈmoᵘrə ˈkɛⁱkt ˈpɛⁱnsənt ˈfoᵘr ˈsɪχ ˈœʸt
ˈsɔmz ˈdɛŋk ˈɪk ˈmɛt ˈhɛ ˈmeⁱ ˈɛn ˈdɑm bəˈlɑndə wə ˈɪŋ ˈχɑrtərhɔᵘsə
ˈwɑn ˈdɑr ˈɪs ˈhɛⁱ ət χəˈlœkɪχst χəˈweⁱst ˈtɔː ˈhɛⁱ ˈɪn də ˈsɛl ˈfɑn də ˈtoᵘwər ˈkwɑm
ˈdɑrˌnɑst ˈhɑŋt ən ˈseⁱr ˈkløʸˌrɛⁱk ɑpˈstrɑkt sχɪldəˈrɛⁱ ˈfɑn əŋ ˈχudə ˈfrɪnt ˈfɑ ˈmeⁱ
ət ˈɪ ˈsoᵘ œʸdˈbœndəχ ˈdɑt əː ˈfeⁱl pləˈsiᵊr ˈɪn ˈsɪχˌsɛləf ˈmut ˈhɛbə
ˈsoᵘˌwɑlz ˈdi ˈfrɪn ˈdɑ ˈtrɔᵘwəns ˈoᵘk ˈɑltɛⁱt ˈheⁱft
ˈɑn də ˈχrɛⁱsə ˈmyᵊr ˈhɑŋt əŋ ˈfoᵘroᵘrlɔχsə ˈteⁱkənɪŋ ˈfɑn ən ˈstrɑtjə ˈɪn
ˈɑmstərdɑm ˈɔᵘt ˈwɛst
ˈdoᵘr ˈdɑt ˈstrɑtjə ˈχɪŋ ˈɪk ˈsɛs ˈjɑr ˈlɑŋ ˈnɑr də ˈlɑχərə ˈsχoᵘl
ˈdɑχələk ˈseⁱr χəˈlœkəχ
ət ˈwɑz də ˈmoᵘjstə ˈtɛⁱt ˈfɑ ˈmeⁱn ˈleⁱfə ɔmˈdɑt ət ˈleⁱrə ˈtɔt ˈnɪdz ˈdɪndə
ˈɔb də ˈfoᵘrˌχrɔnt ˈloᵘpt əm ˈbɑkərcə ˈɑχtər əŋ ˈkɑr
ˈɪk ˈhɛp ˈhɛm χəˈkɛnt
ˈhɛⁱ ˈɪs ˈɪn əm ˈplɑs ˈχrɑŋχəˌstoᵘktə jəˈneⁱfər fərˈdrɔŋkə
ˈɔp əd byᵊˈroᵘ ˈlɪɣd ˈbɛⁱnɑ ˈɔb də ˈtɑfəl ˈɪn ət χəˈheⁱl ˈnɪts
ˈɪk ˈhɔᵘt ˈnɪt ˈfɑm pɑˈpiᵊr
ˈɑn ət ɔˈbɔᵘwə ˈfɑn ən ˈrɛⁱk χəˈfulsˌleⁱfə ˈwɛrəg də ˈkɑmər ˈnɪt ˈmeⁱ
ˈhɛⁱ ˈɪs ən ɔmpərˈsoᵘnləkə ˈrœʸmtə ˈkɔᵘt ˈsɛləfs ˈsoᵘˌwɑls əm pyˈblɪkə ˈrœʸmtə ˈpɑst
əm prɪˈfeⁱ ˈkɑmər fərˈɛⁱst ˈɑfslœʸtɪŋ ˈmɑr ˈsloᵘtə ˈseⁱn ˈɪn ˈdɪt χəˈbɔᵘw fərˈboᵘdə

EMACS (Fundamental) fensnoord.out >walterd>old>grafon SYM1: (2) Font: A (PHONEME) * [More

'hɔᵘt 'nɪt 'fɑ 'mɛᶦŋ 'kɑmər 'mɑr 'ɪk 'hɛp ər 'drɪ 'weᶦkə fɑŋ χə'hɔᵘdə
'dɔndərdɑχ 'ɑχ,tɪn ɑ'prɪl 'smɪdɑχs 'ɔm 'drɪ 'yᵊr
vɑmə ər 'tweᶦ 'jɔŋə,mɑnə 'mɛᶦŋ 'kɑmər 'bɪnə
ɛt ən 'sɪn 'dɪ 'ɪk 'feᶦrtəχ 'jɑr χə'leᶦdə 'foᵘr ət 'lɑtst 'hɑt χə'hoᵘrt
t χə'bɔᵘw 'ɪz bə'sɛt 'y 'dɪn 'deᶦsə 'kɑmər 'bɪnə ən 'hɑləf 'yᵊr tə fər'lɑtə
bə'χreᶦb də ɔŋweᶦr'stambarə 'krɑχt 'fɑn də 'deᶦmoᵘkrɑtsɪ
ɪ ən 'yᵊr 'lɑtər 'stɔnt 'ɪg 'bœʸtə də 'ɑl 'soᵘ 'leᶦχə 'kɑmər 'lɪt 'ɪk 'leᶦχər 'ɑχtər
.n,ɑv də 'fɔlχəndə 'dɑχ 'stɔn də 'kɑmər 'drɪ 'weᶦkə 'ɪ 'mɛᶦn 'hoᵘft
'sɑt 'ɑn də 'ɑndərə 'kɑnt 'fɑn ət 'lɑnt 'wɑt 'mɛᶦm bə'trɔkənheᶦt mɪ'sχɪn
oᵘtər 'mɑktə 'ɪk 'sɑχ 'foᵘr,dyᵊrən də 'kɑmər 'leᶦχ 'ɑχtərχə,lɑtə 'ɔp ən 'doᵘt,stɪlə
.ŋ 'ɛn 'ɪk 'mɑktə ər 'ɪn də 'χeᶦst 'steᶦts ən 'rɛᶦz 'doᵘr 'heᶦn
'χrɛᶦsə 'myᵊr də 'wɪtə 'myᵊr 'sɔᵘdə sə 'ɛlkar 'kœnə 'ɔndər,hɔᵘdə
ɪ də 'neᶦχəndə 'dɑχ 'sɑχ 'ɪk 'ɔb də 'wɪtə 'myᵊr əm 'bøʸrsə 'plɛk
'stɑn 'doᵘr də 'teᶦχə də 'myᵊr 'slɑndə 'døʸr 'steᶦŋ 'kɑŋ fər'teᶦdərə
'lɪχt 'wɑs 'œʸt nɑ'tyᵊrlək
ᵘ 'moᵘrə 'nɔχ 'ɪt 'sɪn
ɛ də 'lɑmpə 'ɑn 'kɑn 'hɛᶦ 'ɪn ət 'froᵘləkə sχɪldə'rɛᶦ 'kɛᶦkə
ːᶦ 'sɪt 'ny 'slɛχs 'χrɛᶦsheᶦt 'ɛn 'sɛᶦm 'pɛᶦnsə 'ɪs 'oᵘfərχə,χɑn 'ɪn ən 'lɪχtə 'meᶦlɑŋɣoᵘlɪ
'bɑkər 'dœwt 'sɛᶦŋ 'kɑrəcə ə 'dœʸstər 'ɪn 'ɛ 'nɪt 'lɑŋər 'ɪn də 'rɪχtɪŋ
ɪn də 'χloᵘrɪsə 'brœχ 'wɑrɑχtər 'mɛᶦn 'sχoᵘ 'lɑχ
ɪ də 'plɑntə 'ɪ 'kreᶦχ 'meᶦdə,leᶦdə 'mɛt 'hɛn
ən 'ɪs əŋ 'wɑt ɔ'noᵘsələ 'heᶦ 'lɛᶦkt 'oᵘk 'nɪ tə 'χrujən
ːᶦ 'sɑl 'fɑn də 'leᶦχtə 'nɪdz bə'χrɛᶦpə 'mɑr də 'ɑndərə 'dɪ əm 'boᵘm proᵘ'beᶦr tə 'sɛᶦn
. 'mut 'lɛᶦdə 'ɔndər də 'stɪltə 'fɑn də 'doᵘt 'dɪ 'ɛlkə bə'sɛtɪŋ 'meᶦ,brɛŋt
'trɑχtə də 'kɑmər tə bə'woᵘnə 'ɪk 'χɪŋ 'ɑχtər ət 'leᶦχə byᵊ'roᵘ 'sɪtə
ɪ 'keᶦk 'nɑr də 'kɑstə sə 'kœnə 'sɪɣ 'beᶦsɪχ,hɔᵘdə 'mɛ də 'bukə 'dɪ 'ɛr,ɪn 'stɑn
'χɑf sə 'tu 'woᵘrdəm,bukə 'ɛ nɑ'slɑχ,werkə 'seᶦ 'mujlək tə 'leᶦsə 'beᶦ 'sleχt 'lɪχt
ɪ ɔp'wɪndənt 'sɛᶦn sə 'oᵘk 'nɪt əŋ 'kɔn 'sɪχ fər'makə 'mɛt 'jar,χɑŋən
ɪŋ 'feᶦlə 'teᶦt,sχrɪftə 'mɑr də 'dɛrdə 'dɪ 'hɑt 'nɪt 'feᶦl 'ɑndərs
ɪn 'ɑn,sɛtə 'fɑn 'skrɪpsɪs 'ɛn 'dɑr 'kɔm jə 'leᶦχə 'weᶦkə 'nɪt 'meᶦ 'doᵘr
'teᶦləfoᵘn 'slɑpt 'ɔb də 'lɛsənar 'dɑχt 'ɪk 'ɔb də 'twɑləvdə 'dɑχ
ɑnt poᵘwə'sɪ 'kɑ mɪ'sχɪn ən 'leᶦχtə 'wɑt 'fœlə
ɻ 'brɛŋ 'ɪk ən 'teᶦkə 'fɑn 'leᶦfə 'oᵘfər 'nɑr 'χɪns 'nɑr də 'plɑntə də sχɪldə'reᶦjə
'kɑstə 'ɛn də 'tɑfəls 'ɪk 'hɛp 'tweᶦ 'keᶦr ət 'nœmər 'fɑ 'mɛᶦŋ 'kɑmər χə'bɛlt
ɪt 'ɪn də 'ɑfɔnt wɑ'neᶦr də fər'lɑtənheᶦt ət 'χroᵘtst 'mut 'sɛᶦŋ χə'weᶦst
'hoᵘptə 'dɑ də 'døʸr 'sɔᵘ 'oᵘpən,stɑn 'ɛn də 'bɛl 'doᵘr də 'heᶦlə 'χɑŋ tə 'hoᵘrə
ˌᵘ 'sɛᶦn
kɔ'leᶦχɑ 'mɔχt 'ɔb də 'feᶦf,tɪndə 'dɑχ 'fɑn də bə'sɛtərs 'ɪts 'fɑn 'sɛᶦŋ 'kɑmər 'hɑlə
ɪdər χə'leᶦdə 'heᶦ bə'woᵘn də 'kɑmər 'nɑz də 'mɛᶦnə

'fɔlχəndə 'dɑɣ 'bɛldə 'hɛˑ 'mɛˑ jə 'kamər 'stat 'ɛr,bɛˑ 'soᵘ,walꜱ jə 'hɛm 'hɛpt
r'latə
ɪ 'dœrəvdə 'niˌ tə 'fraχə 'nar də 'tu,stant 'fan də 'møʸbəls də 'plantə
oᵘrə 'ɛn ət 'stratjə
'kløʸ,rɛˑkə sχɪldə'rɛˑ 'χaf 'mɛˑ 'χeˑn 'sɔrχə 'diˌ 'heˑft χə'nuχ 'an 'seˑn 'ɛˑχə χə'lœk
ant 'ɪg 'druχ 'mɛˑn 'leˑt 'ɪn 'stɪltə
ə 'driˌ 'weˑkə 'kɛndə 'ɪk 'fan də 'kamər 'ɛlkə 'sɛntsiˌ,meˑtər
ɪ 'ɛlkə 'flɛk 'ɪn də 'fluᵊrbə,kleˑdɪŋ 'ɛn 'ɔp ə 'tafəl,blat
ɔ 'mandaɣ 'dɛrtiˌ 'mɛˑ 'kwam ər ən 'ɛˑndə 'an də bə'sɛtɪŋ
n 'halaf 'twaləf 'ɪn də 'mɔrχə 'kwam 'ɪg də 'kamər 'bɪnə
ˌəs 'hat 'sɪχ 'weˑr 'ɔp 'sɪχ,seləf tə'rœχə,trɔkə də 'leˑχtə 'was 'weˑr
t 'seˑn oᵘr'sprɔŋkələk χə'dantə χə'fœlt
ɪ də 'huk 'aχtər əŋ 'kast 'fɔnt 'ɪk əm para'pl̩y 'diˌ 'hat 'ɪk 'heˑləmal fər'χeˑtə
ɛt 'hɛm 'hat 'ɪk 'n̩y ət 'χroᵘtstə 'meˑdə,leˑdə 'ɪg 'bɛ 'mɛˑn 'sχœltχə,fuləns
ˑˑχən,oᵘfər 'hɛm 'nɔχ 'niˌt 'kweˑt
ɔ 'foᵘr,maliχə bə'sɛtər 'kɔmt 'fraχə 'nar də 'datœm 'fan əŋ kɔ'leˑχə
ɪt 'hɛˑ də 'kamər 'diˌ 'hɛˑ 'soᵘ 'deˑt 'leˑdə 'soᵘ 'mar 'ɪn 'dœrəvdə
ˑˑft 'mɛˑ ət 'meˑst fər'beˑstərt

ˑˑs 'fɛns

; (Fundamental) fensnoord.out >walterd>old>grafon SYM1: (2) Font: A (PHONEME) * [More
18:28 Hardcopy of Zmacs Frame 1 has been bitmapped to AGFA]

APPENDIX A.8

A.8 Sample Phonological Rule Applications

For each rule, a short verbal description is given, and some examples with the solution computed by GRAFON and a derivation for each syllable.

FINAL-DEVOICING

Voiced obstruents are devoiced in the coda of syllables. The effect of this rule may be undone by assimilation rules following it. E.g.: bezorgd (worried), zoogdier (mammal), word (become).

bezorgd |bə'zɔrəɣt|
"zorgd"
(1. FINAL-DEVOICING)
(2. SCHWA-INSERTION)

zoogdier |'zoɣˌdi:r|
"zoog"
(1. FINAL-DEVOICING)
(2. REGRESSIVE-ASSIMILATION)
"dier"
(3. VOWEL-DIFTONGISATION-1)

word |'wɔrt|
"word"
(1. FINAL-DEVOICING)

INITIAL-DEVOICING

In our opinion, this rule is restricted to northern variants of standard Dutch. Voiced fricatives (/ɣ/ /z/ and /v/) become voiceless in syllable-initial position. Initial devoicing must apply before intervocalic voicing because the latter can undo the effect of the former. E.g.: zegevuur (victory fire), meegaand (compliant).

zegevuur |'seˈχəˌfyˑr|
"ze"
(1. INITIAL-DEVOICING)
(2. VOWEL-DIFTONGISATION-2)
"ge"
(3. INITIAL-DEVOICING)
"vuur"
(4. INITIAL-DEVOICING)
(5. VOWEL-DIFTONGISATION-1)

meegaand |'meˑˌɣantˌ|

"mee"
(1. VOWEL-DIFTONGISATION-2)
"gaand"
(2. FINAL-DEVOICING)
(3. INITIAL-DEVOICING)
(4. INTERVOCALIC-VOICING)

INTERVOCALIC-VOICING

Voiceless fricatives between stressable vowels (all vowels except /ə/) or liquids are voiced. Domain of this rule is the word (operationalized as the syllable string between two internal, two external, or an internal and an external word boundary). E.g.: televisie (television), basis (base), Israël, mensa.

televisie |'teləvɪziˌ|

"sie"
(1. INTERVOCALIC-VOICING)

basis |'bazɪsˌ|

"sis"
(1. INTERVOCALIC-VOICING)

Israël |'ɪzraˌʔɛlˌ|

"is"
(1. INTERVOCALIC-VOICING)
"ra"
(2. HIATUS-FILLING)

mensa |'menzaˌ|

"sa"
(1. INTERVOCALIC-VOICING)

PLOSIVE-TO-FRICATIVE

In some cases, plosive /t/ is pronounced /s/ or /ts/. The /t/ must occur word-internally in a variant of the affix -tie. After a vowel, /n/ or /r/, /t/ becomes /s/ in the southern dialects, and /ts/ in the northern dialects. After /s/, /t/ remains /t/, and after other consonants /t/ becomes /s/ in both dialects. It must be ordered after intervocalic voicing to avoid the voicing of the /s/. E.g.: suggestie (suggestion), administratie (administration), actie (action), politie (police).

suggestie |sœ'ɣɛstɪ|

actie |'ɑksɪ|

"tie"

(1. PLOSIVE-TO-FRICATIVE)

administratie |ɑtmɪnɪ'strasɪ|

"ad"

(1. FINAL-DEVOICING)

"tie"

(2. PLOSIVE-TO-FRICATIVE)

politie |poʷ'lɪtsɪ|

"po"

(1. VOWEL-DIFTONGISATION-2)

"tie"

(2. PLOSIVE-TO-FRICATIVE)

N-DELETION

If a syllable-rhyme is equal to /ə n/, /n/ can be deleted if an internal or external word boundary follows. One restriction on the rule cannot be integrated in the system without syntactic knowledge: the inflected forms ending in /ən/ of verbs with an infinitive ending in /ənən/, resist n-deletion. E.g. ik open (I open) --> /Ik opə n/. Examples: lopen (run), opendoen (make open), houten (wooden).

lopen |'lopə|
"pen"
(1. N-DELETION)

opendoen |'opə,dun|
"pen"
(1. N-DELETION)

houten |'hɔʷtə|
"ten"
(1. N-DELETION)

VOWEL-DIPHTHONGIZATION-1

The name of this rule refers to the shifting of /i/ /y/ and /u/ in the direction of /ə/ before /r/ in northern Dutch. In southern Dutch, these vowels are lengthened in that position, but not diphthongized. E.g.: muur (wall), voert (feeds), wierp (threw)

muur |'my:r|
"muur"
(1. VOWEL-DIFTONGISATION-1)

voert |'vu:rt|
"voert"
(1. VOWEL-DIFTONGISATION-1)

wierp |'wi:rəp|
"wierp"
(1. VOWEL-DIFTONGISATION-1)
(2. SCHWA-INSERTION)

muur |'myᵊr|
"muur"
(1. VOWEL-DIFTONGISATION-1)

voert |'fuᵊrt|
"voert"
(1. INITIAL-DEVOICING)
(2. VOWEL-DIFTONGISATION-1)

VOWEL-DIPHTHONGIZATION-2

This rule is restricted to northern standard Dutch. High vowels (/ø/, /o/ and /e/) are diphthongized into the direction of /y/,/u/ and /i/ respectively, if they are not followed by /r/. E.g.: deuk (dent), zo (so), thee (tea).

zo |'soᵘ|

"zo"

(1. INITIAL-DEVOICING)

(2. VOWEL-DIFTONGISATION-2)

thee |'teᶦ|

"thee"

(1. VOWEL-DIFTONGISATION-2)

deuk |'døʸk|

"deuk"

(1. VOWEL-DIFTONGISATION-2)

HIATUS-FILLING

Whenever two vowels follow each other immediately (i.e. when the coda of one syllable is empty and the onset of the next, too), the transition is filled by /j/ after front vowels or diphthongs, by /w/ after back vowels or diphthongs, and by a /ʔ/ after /a/, /ɑ/ and /ə/. E.g.: duo, Israel, zeeën (seas), beamen (confirm).

duo |'dywo|

"du"
(1. HIATUS-FILLING)

Israël |'ɪzrɑ,ʔɛl|

"is"
(1. INTERVOCALIC-VOICING)
"ra"
(2. HIATUS-FILLING)

zeeën |'zejə|

"zee"
(1. HIATUS-FILLING)
"en"
(2. N-DELETION)

beamen |bə'ʔaмə|

"be"
(1. HIATUS-FILLING)
"men"
(2. N-DELETION)

SCHWA-INSERTION

If in the coda of a syllable a liquid is followed directly by a bilabial, labiodental or velar consonant, a schwa is inserted between them. E.g.: wierp (threw), wurgt (strangles), alp, herfst (autumn).

wierp |'wɫ:rəp|

"wierp"
(1. VOWEL-DIFTONGISATION-1)
(2. SCHWA-INSERTION)

wurgt |'wœrəχt|

"wurgt"
(1. FINAL-DEVOICING)
(2. SCHWA-INSERTION)

alp |'ɑɫəp|

"alp"
(1. SCHWA-INSERTION)

herfst |'hɛrəfst|

"herfst"
(1. SCHWA-INSERTION)

CLUSTER-REDUCTION

In some cases, /t/ is deleted when surrounded by consonants. At present the domain of the rule is defined to be the word; the rule is blocked over external word boundaries. E.g.: herfststorm (autumnal tempest), vondst (discovery), lichts (light), vruchtbaar (fertile), kastje (little cupboard).

herfststorm |'hɛrəf,stɔrəm|

"herfst"
(1. SCHWA-INSERTION)
(2. CLUSTER-REDUCTION)
(3. DEGEMINATION)
"storm"
(4. SCHWA-INSERTION)

vondst |'vɔnst|

"vondst"
(1. FINAL-DEVOICING)
(2. CLUSTER-REDUCTION)

lichtst |lɪχst|

"lichtst"
(1. CLUSTER-REDUCTION)

vruchtbaar |'vrœɣ,bar|

"vrucht"
(1. CLUSTER-REDUCTION)
(2. REGRESSIVE-ASSIMILATION)

kastje |'kɑʃə|

"kast"
(1. CLUSTER-REDUCTION)
"je"
(2. PALATALISATION)

PALATALIZATION

The combination of /t/ and semi-vowel /j/ is palatalized to palato-alveolar plosive /c/. The combination of /s/ with /j/ results in a palato-alveolar fricative /ʃ/. This assimilation process occurs mainly in Dutch diminutives. E.g.: mandje (little basket), meisje (girl), matje (little mat), tasje (little bag).

mandje	|'mɑɲcə|
"mand"	
(1. FINAL-DEVOICING)	
(3. NASAL-ASSIMILATION)	
"je"	
(2. PALATALISATION)	
meisje	|'mɛiʃə|
"je"	
(1. PALATALISATION)	
matje	|'mɑcə|
"je"	
(1. PALATALISATION)	
tasje	|'tɑʃə|
"je"	
(1. PALATALISATION)	

PROGRESSIVE-ASSIMILATION

Voiced fricatives in the onset of a syllable are made voiceless, if the last phoneme in the coda of the preceding syllable is a voiceless obstruent. This rule also applies across word-boundaries (it is restricted by the phonological phrase). This rule never applies in the northern variant of Dutch because initial-devoicing is ordered before it. E.g.: de kat zit (the cat sits), loopgraaf (trench), AKZO (company name).

de kat zit |də 'kɑt 'sɪt|

"kat"

(1. PROGRESIVE-ASSIMILATION)

loopgraaf |'lop,xraf|

"loop"

(1. PROGRESIVE-ASSIMILATION)

AKZO |'ɑksɔ|

"ak"

(1. PROGRESIVE-ASSIMILATION)

REGRESSIVE-ASSIMILATION

Voiceless obstruents in the coda of a syllable are made voiced if the first phoneme in the onset of the following syllable is a voiced plosive. Again, the rule applies across word boundaries and is restricted by the phonological phrase. E.g.: liedboek (song-book), zakdoek (handkerchief), liefde (love) , herfstdag (day in autumn).

liedboek |'lɪd,buk|

"lied"

(1. FINAL-DEVOICING)

(2. REGRESSIVE-ASSIMILATION)

zakdoek |'zɑg,duk|

"zak"

(1. REGRESSIVE-ASSIMILATION)

liefde |'lɪvdə|

"lief"

(1. REGRESSIVE-ASSIMILATION)

herfstdag |'herəvz,dɑx|

"herfst"

(1. SCHWA-INSERTION)

(2. CLUSTER-REDUCTION)

(3. REGRESSIVE-ASSIMILATION)

"dag"

(4. FINAL-DEVOICING)

NASAL-ASSIMILATION

The alveolar nasal /n/ assimilates the place of articulation of the following consonant, within and across word boundaries. E.g.: onbepaald (indefinite), ongewoon (unusual), onjuist (incorrect), onvrij (not free), zink (zinc), mandje (little basket).

onbepaald |ɔmbə'palt|

"on"
(1. NASAL-ASSIMILATION)
"paald"
(2. FINAL-DEVOICING)

ongewoon |ɔŋɣə'won|

"on"
(1. NASAL-ASSIMILATION)

onjuist |ɔɲ'jœʏst|

"on"
(1. NASAL-ASSIMILATION)

onvrij |ɔɱ'vrɛɪ̯|

"on"
(1. NASAL-ASSIMILATION)

zink |'zɪŋk|

"zink"
(1. NASAL-ASSIMILATION)

mandje |'mɑɲcə|

"mand"
(1. FINAL-DEVOICING)
(3. NASAL-ASSIMILATION)
"je"
(2. PALATALISATION)

DEGEMINATION

Whenever two identical consonants follow each other immediately, the cluster is reduced to one consonant, within and across word boundaries. E.g.: in mijn (in my), postzegel (stamp).

in mijn |ˈɪ ˈmɛɪn|

"in"
(1. NASAL-ASSIMILATION)
(2. DEGEMINATION)

postzegel |ˈpɔˌsɛɣəl|

"post"
(1. CLUSTER-REDUCTION)
(2. PROGRESIVE-ASSIMILATION)
(3. DEGEMINATION)

REFERENCES

Adriaens, G. *Process Linguistics: the theory and practice of a cognitive-scientific approach to natural language understanding.* Dissertation Katholieke Universiteit Leuven, 1986.

Allen, J. 'Speech Synthesis from Text.' In: Simon, J.C. (Ed.) *Spoken Language Generation and Understanding.* Dordrecht: Reidel, 1980, p. 3 - 39.

Anderson, J.R. and Reiser, B.J. 'The Lisp Tutor.' *Byte*, 10 (4), 1985.

Angell, R.C., G.E. Freund and P.Willett. 'Automatic spelling correction using a trigram similarity measure.' *Information Processing and Management,* 19.4 (1983), 255-261.

Aronoff, M. *Word Formation in Generative Grammar.* Cambridge Mass.: MIT-Press, 1976.

Aronoff, M. 'Lexical representations.' In D. Farkas, W.M. Jacobsen, K.W. Todrys (eds.) *Papers from the parasession on the lexicon.* Chicago, 1978.

Assink, E.M.H. *Leerprocessen bij het spellen.* Diss. Rijksuniversiteit Utrecht, 1983.

Assink, E.M.H. 'Het in kaart brengen van spellingproblemen.' *Tijdschrift voor spellingbeheersing* 6, 1984, 264-276.

Bakel, J. van, *Fonologie van het Nederlands.* Utrecht: Bohn, Holkema en Scheltema, 1976.

Bakel, J. van, 'Methodologie van de computerlinguistiek.' *Gramma* 7, 1983, 175-188.

Bakker, J.J.M. *Constant en Variabel, de fonematische structuur van de Nederlandse woordvorm.* Assen, 1971.

Ballard, D.H. 'Cortical connections and parallel processing: structure and function' *The Behavioral and Brain Sciences* 9, 1986, 67-120.

Bartlett, E.J. 'Learning to Revise: Some Component Processes.' In: M. Nystrand (Ed.). *What Writers Know.* New York: Academic Press, 1982, p. 345-363.

Barton, G.E. 'The Computational Complexity of Two-level Morphology.' MIT AI-Memo 856, 1985.

Berg, B. van den, *Foniek van het Nederlands.* Den Haag: Van Goor, 1972.

Berkel, B. Van, *Spelterapuit: een algoritme voor spel- en typefoutcorrectie gebaseerd op grafeem-foneemomzetting.* MA-thesis, Department of Psychology, University of Nijmegen, 1986.

Berko, J. 'The child's learning of English morphology.' *Word* 14, 1958, 1590-177.

Bobrow, D.G. and T. Winograd. 'An overview of KRL, a knowledge representation language.' *Cognitive Science*. 1(1), 1977, 3-46.

Booij, G.E., *Dutch Morphology*. Dordrecht: Foris Publications, 1977.

Booij, G.E. (ed.) *Morfologie van het Nederlands*. Amsterdam: Huis aan de drie grachten, 1979.

Booij, G.E., *Generatieve fonologie van het Nederlands*. Utrecht: Spectrum, 1981.

Booij, G.E. 'Lexical Phonology, Final Devoicing and Subject Pronouns in Dutch.' In Bennis, H. and F. Beukema (eds.) *Linguistics in the Netherlands*. Dordrecht: Foris Publications, 1985.

Booij, G.E., C. Hamans, G. Verhoeven, F. Balk, Ch. H. van Minnen. *Spelling*. Groningen: Wolters-Noordhoff, 1979.

Boot. M., *Taal, tekst, computer*. Katwijk: Servire, 1984.

Brandt Corstius, H. *Exercises in Computational Linguistics*. Amsterdam: Mathematical Centre Tracts 30, 1970.

Brandt Corstius, H. 'Wat is computer-taalkunde?' Inaugurale rede, University Press Rotterdam, 1974.

Brandt Corstius, H. *Computer/taalkunde*. Muiderberg: Coutinho, 1978.

Brandt Corstius, H. 'Weg met de Computer!'. *Kennis en methode* 1, 1981.

Breuker, J. and S. Cerri. 'A New Generation of Teaching Machines: intelligent and rather-intelligent computer assisted instruction discussed and exemplified.' In: E. Van Hees and A. Dirkzwager (Eds.) *Onderwijs en de nieuwe media*. Lisse: Smets & Zeitlinger, 1982.

Brown, J.S. and Burton, R.R. 'Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science* 2, 1978, p. 155-192.

Butterworth, B. 'Lexical Representation.' In: B. Butterworth (ed.) *Language Production*, vol 2, London: Academic Press, 1983, 257-294.

Byrd, J.R. (1983) 'Word Formation in Natural Language Processing Systems.' Proceedings IJCAI-83, 704-706.

Cannon, H.I. 'A non-hierarchical approach to object-oriented programming.' Cambridge Mass.: MIT AI-Lab, 1982.

Charniak, E. and D. McDermott. *Introduction to Artificial Intelligence*. Reading, Mass.: Addison-Wesley, 1985.

Cherry, L. 'A Toolbox for Writers and Editors.' In *Proceedings Office Automation Conference*, 1981.

Chomsky, N. *Aspects of the theory of syntax*. Cambridge Mass.: MIT-Press, 1965.

Chomsky, N. 'Remarks on Nominalisation.' In Jacobs, R. and P. Rosenbaum (eds.) *Readings in English Transformational Grammar*, 1970.

Chomsky, N. *Lectures on Government and Binding*. Dordrecht: Foris, 1981.

Chomsky, N. *The Generative Entreprise. A Discussion with Riny Huybregts and Henk van Riemsdijk*. Dordrecht: Foris Publications, 1982.

Chudacek, J. 'Statistische en combinatorische eigenschappen van trigrammen.' IWIS-TNO rapport (1983), Den Haag.

Chudacek, J. en C.A. Benschop. 'Reconstructie van tekstdelen uit hun syntactische sporen.' IWIS-TNO rapport (1981) Den Haag.

Cohen, A. *Fonologie van het Nederlands en het Fries*. Den Haag: Nijhoff, 1961.

Collier, R., *Nederlandse Grammatica-1*. Leuven: Acco, 1981.

Collier, R. and F.G. Droste, *Fonetiek en Fonologie*. Leuven: Acco, 1977.

Collins, A. and D. Gentner. 'A Framework for a Cognitive Theory of Writing.' In: Gregg and Sternberg (eds.) *Cognitive Processes in Writing*. New Jersey: Lawrence Erlbaum Ass., 1980.

Cutler, A., J.A. Hawkins, G. Gilligan. 'The suffixing preference: a processing explanation.' *Linguistics* 23 (1985), 723-758.

Daelemans, W. 'Automatische detectie en correctie van spellingfouten.' Internal Report University of Nijmegen, 84FU10, 1984.

Daelemans, W., 'Automatic Spelling Error Detection with an Unlimited Vocabulary for Dutch.' Memo, University of Nijmegen, 1985a.

Daelemans, W. 'GRAFON: An Object-oriented System for Automatic Grapheme to Phoneme Transliteration and Phonological Rule Testing.' Memo, University of Nijmegen, 1985b.

Daelemans, W. 'Two syllabification algorithms for Dutch.' Memo, University of Nijmegen, 1985c.

Daelemans, W. 'Dutch verbal inflections: an object-oriented implementation.' Memo, University of Nijmegen, 1986.

Daelemans, W., D. Bakker and H. Schotel. 'Automatische detectie en correctie van spelfouten.' *Informatie*, 26(12), (1984), 952-956.

Daelemans, W., G. Kempen and L. Konst. 'Natural Language Facilities for the Intelligent Office Workstation.' Interim report of ESPRIT project OS-82, 1985.

Damerau, F.J. 'A technique for computer detection and correction of spelling errors.' *Communications of the ACM*, 7(3),(1964), 171-176.

Desain, P. 'TREE DOCTOR: A software package for graphical manipulation and animation of tree structures.' K.U.Nijmegen, Psychology Department, Internal Report 86 CW 01, 1986.

Domenig, M. and P. Shann (1986) 'Towards a Dedicated Database Management System for Dictionaries.' Proceedings COLING-86, 91-96.

Droste, F.G. *Vertalen met de computer. Mogelijkheden en moeilijkheden.*Groningen: Wolters-Noordhoff, 1966.

Droste, F.G. 'Reflections on metalanguage and object-language.' *Linguistics* 21, 1983, 675-699.

Droste, F.G. 'Language, Thought and Mental Models.' Semiotica 56, 31-98.

Droste, F.G. (Ed.) *Stromingen in de hedendaagse linguistiek.* Leuven: Universitaire Pers Leuven, 1985.

Eynde, F. Van. *Betekenis, vertaalbaarheid en automatische vertaling.* Dissertatie Katholieke Universiteit Leuven, 1985.

Faulk, R.D. 'An inductive approach to language translation.' *Communications of the ACM,* 7 (1964), 647.

Flesch, R. *Helder schrijven, spreken, denken.* Van Loghum/Slaterus, 1976.

Fodor, J.A. *The Modularity of Mind.* Cambridge, Mass.: MIT Press, 1983.

Friedman, J. *A computer model of transformational grammar.* New York: Elsevier, 1971.

Fromkin, V. (Ed.). *Speech Errors as Linguistic Evidence.* The Hague: Mouton, 1973.

Fromkin, V.A. and D.L. Rice, 'An interactive phonological rule testing system'. In: Coling, *Preprints of the International Conference on Computational Linguistics,* 1969.

Garrett, M.F. 'The analysis of sentence production.' In: Bower, G.H. (Ed.) *The psychology of learning and motivation. Vol 9.* New York: Academic Press, 1975, 133-177.

Gazdar, G. 'NLs, CFLs and CF-PSGs.' In: Sparck Jones, K. and Y. Wilks (Eds.) *Automatic Natural Language Parsing*. New York: Wiley, 1983.

Gazdar, G. 'Computational Tools for Doing Linguistics: Introduction'. *Linguistics* 23, 1985, 185-187.

Gazdar, G. 'Finite State Morphology.' *Linguistics* 23, 1985, 597-607.

Gebruers, R. 'Het vertaalsysteem Metal' In: *Automatische vertaling aan de K.U. Leuven*. Leuven: Acco, 1986.

Geeraerts, D. and G. Janssens. *Wegwijs in Woordenboeken*. Assen: van Gorcum, 1982.

Geerts, G. et al. (eds.), *Algemene Nederlandse Spraakkunst*. Groningen: Wolters-Noordhoff, 1984.

Gibson, E.J. and L. Guinet. 'Perception of inflections in brief visual presentation of words.' *Journal of verbal learning and verbal behavior* 10, 1971, 182-189.

Glencross, Courner, Nilsson. The Flinders typing Project. Report (1979).

Goldberg, A. and D. Robson. *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley, 1983.

Golding, A.R. *Lexicrunch: an expert system for word morphology*. Unpublished M.Phil thesis, Edinburgh University, 1984.

Golding, A.R. and H.S. Thompson. 'A morphology component for language programs.' *Linguistics* 23, 1985, 263-284.

Gosling, J. *Unix Emacs*. Carnegie Mellon University. 1981.

Greanias, E.C., 'Computer aids for Spelling Correction and Word Selection.' Paper presented at IBM-Europe Institute, Davos, 1984.

Grudin, J. T. 'Error Patterns in Novice and Skilled Transcription Typing.' In: Cooper, W. E. (ed.) *Cognitive Aspects of Skilled Typewriting*. New York: Springer Verlag, 1983.

Halle, M. 'Prolegomena to a Theory of Word Formation' In *Linguistic Inquiry* 4, 1973, 3-16.

Hayes, J. and L.S. Flower. 'Identifying the Organization of Writing Processes.' In: Gregg and Steinberg (Eds.). *Cognitive Processes in Writing*. Hillsdale (N.J.): Erlbaum, 1980, p.3-30.

Hays, D.G. *An Introduction to Computational Linguistics*. New York: American Elsevier, 1967.

Heer, T. De. 'De informatiesporenmethode.' IWIS-TNO rapport (1982), Den Haag.

Henderson, L. 'Toward a psychology of morphemes.' In A.W. Ellis (ed). *Progress in the psychology of language, vol. I.* London: Erlbaum.

Hendrix, G.G. 'Encoding Knowledge in Partitioned Networks.', In: Findler, N.V. (Ed.) *Associative Networks - Representation and Use of Knowledge by Computers.* Academic Press, 1979.

Heuven, V. van. *Spelling en lezen. Hoe tragisch zijn de werkwoordsvormen?.* Assen: Van Gorcum, 1978.

Hewitt, C, P. Bishop and R. Steiger. 'A Universal, Modular Actor Formalism For Artificial Intelligence.' IJCAI, 1973.

Hill, D.R. 'Spoken Language Generation and Understanding: A Problem and Application Oriented Overview.' In: Simon, J.C. (Ed.) *Spoken Language Generation and Understanding.* Dordrecht: Reidel, 1980, p. 3 - 39.

Hoenkamp, E. *Een computermodel van de spreker: Psychologische en Linguistische Aspecten.* Unpublished dissertation, University of Nijmegen.

Hoenkamp, E. 'Weg met de computer?' *Gramma* 9, 1985, 27-45.

Honig, H.J. *Oracle User's Manual.* Delft University of Technology, 1984.

Hudson, R., *Word Grammar.* Oxford: Blackwell, 1984.

Hudson, R., and W. Van Langendonck. 'Word Grammar.' In: Droste, F.G. (Ed.) *Stromingen in de hedendaagse linguistiek.* Leuven: Universitaire Pers Leuven, 1985, 192-229.

Isoda, M., H. Aiso, N. Kamibayashi and Y. Matsunaga (1986) 'Model for Lexical Knowledge Base.' Proceedings COLING-86, 451-453.

Jackendoff, R.S., 'Semantic and Morphological Regularities in the Lexicon.' *Language* 51, 639-71.

Jakimik, J., R.A. Cole and A.I. Rudnicky. 'Sound and spelling in spoken word recognition.' *Journal of memory and language* 24, 1985, 165-178.

Janssen, T.M.V. 'A computer program for Montague grammar.' In: Groenendijk, J. and M. Stokhof (Eds.) *Proceedings of the Amsterdam Colloquium on Montague Grammar and Related Topics.* Amsterdam, 1976.

Johnson, M. 'Computer Aids for Comparative Dictionaries.' *Linguistics* 23, 1985, 285-302.

Johnson-Laird, P.N. 'Mental Models in Cognitive Science.' *Cognitive Science* 4, 1980, 71-115.

Joseph, D.M. and R.L. Wong. 'Correction of Misspellings and Typographical Errors in a Free-text Medical English Information Storage and Retrieval System.' *Methods Inform. Med.*, 18(4), 1979, 228-234.

Kay, A. 'SMALLTALK, A Communication medium for children of all ages.' Palo Alto, California: Xerox Palo Alto Pesearch Center, Learning Research Group, 1974.

Kay, M. 'Functional Grammar.' *Proceedings 5th annual meeting of the Berkeley Linguistic Society*, 1979.

Kay, M. 'When meta-rules are not meta-rules.' In: Sparck Jones, K. and Y. Wilks (Eds.) *Automatic Natural Language Parsing*. New York: Wiley and Sons, 1983, p. 94-116.

Kay, M. 'Parsing in a functional unification grammar.' In Dowty, D., L. Karttunen and A. Zwicky (Eds.) *Natural Language Parsing*. London: Cambridge University Press, 1985, 251-278.

Kempen, G. 'Het Artificiële-intelligentieparadigma.' In: J.G.W. Raaijmakers, P.T.W. Hudson and A.H. Wertheim (reds.) *Metatheoretische aspecten van de psychonomie*. Van Loghum Slaterus, 1983.

Kempen, G. 'Natural Language Technology for Office Workstations.' Report University of Nijmegen, 1984.

Kempen, G., L. Konst, K. De Smedt. 'Taaltechnologie voor het Nederlands.' *Informatie*, 26 (11), 878-881, 1984.

Kempen, G., H. Schotel and F. Pijls. 'Taaltechnologie en Taalonderwijs.' Report, University of Nijmegen, 1984.

Kempen, G., G. Anbeek, P. Desain, L. Konst, K. De Smedt. 'Author Environments: Fifth Generation Text Processors.' In: *ESPRIT '86*. Amsterdam: North-Holland, 1986.

Kempen, G. and E. Hoenkamp. 'An Incremental Procedural Grammar for Sentence Production.' *Cognitive Science*, forthcoming.

Kerkhoff, J., J. Wester and L. Boves, 'A compiler for implementing the linguistic phase of a text-to-speech conversion system'. In: Bennis and Van Lessen Kloeke (eds), *Linguistics in the Netherlands*, p. 111-117, 1984.

Kerstens, J. 'Abstracte Spelling.' *De Nieuwe Taalgids*, 74-1, 1981, 29-44.

Kiparsky, P. 'Metrical Structure Assignment is Cyclic'. *Linguistic Inquiry* 10, 1979, 421-42.

Knuth, D.E. *The Art of Computer Programming. Vol. 3. Sorting and Searching.* Reading: Addison-Wesley, 1973.

Knuth, D.E. *TEX and Metafont: New Directions in Typesetting.* Bedford, MA: Digital Press, 1979.

Konst, L. 'A Syntactic Parser Based on Filtering.' K.U.Nijmegen: memo Psychological Laboratory, 1986.

Konst, L., W. Daelemans and G. Kempen. 'An Author System for an Intelligent Office Workstation.' Final report ESPRIT pilot project OS-82, 1984.

Koskenniemi, K. *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production.* University of Helsinki: Department of General Linguistics, Publication 11, 1983.

Koskenniemi, K. 'Two-level morphology.' Proceedings of COLING 1984.

Kruyskamp, C. *Groot Woordenboek van de Nederlandse Taal.* Utrecht/Antwerpen: Van Dale Lexicografie, 1982.

Kuipers, B. 'A frame for Frames: Representing Knowledge for Recognition.' In: Bobrow and Collins (Eds.) *Representation and Understanding: Studies in Cognitive Science* New York: Academic Press, 1975, p.151-184.

Kwee, Tjoe Liong. 'A Computer Model of Functional Grammar.' In: G. Kempen (Ed.) *Natural Language Generation: Advances in Artificial Intelligence, Psychology and Linguistics.* Dordrecht: Kluwer Academic Publishers (in preparation).

Laudanna, A. and C. Burani. 'Address mechanisms to decomposed lexical entries.' *Linguistics* 23 (1985), 775-792.

Ledbetter, L. and B. Cox. 'Software-ICs.' *Byte*, June 1985.

Lenat, D.B. 'The Nature of Heuristics.' *Artificial Intelligence*, 19, 1982, 189-249.

Lieberman, H. 'A Preview of ACT1.' MIT: Artificial Intelligence Laboratory, Memo No. 625, 1981.

Lyons, J. *Language, Meaning and Context.* Fontana Paperbacks, 1981.

MacKay, D.G. 'On the retrieval and lexical structure of verbs.' *Journal of verbal learning and verbal behavior* 15, 1976, 169-182.

MacKay, D.G. 'The structure of words and syllables: evidence from errors in speech.' *Cognitive Psychology* 3, 1972, 210-227.

Marcke, K. Van. 'FPPD: A Consistency Maintenance System Based on Forward Propagation of Proposition Denials.' AI-Memo 86-1, 1986.

Marcke, K. Van. *The KRS Manual.* Unpublished draft, V.U.B. AI-Lab., 1986.

Marr, D. 'Artificial Intelligence - A Personal View.' *Artificial Intelligence* 9, 1977, 37-48.

Marr, D. *Vision: a computational investigation into the human representation and processing of visual information.* San Fransisco: Freeman, 1982.

Marslen-Wilson, W.D. 'Speech understanding as a psychological process.' In J.C. Simon (Ed.). *Spoken Language Generation and Understanding.* Dordrecht: Reidel, 1980.

Marslen-Wilson, W.D. and A. Welsh. 'Processing interactions and lexical access during word recognition in continuous speech.' *Cognitive Psychology* 10, 1978, 29-63.

Marslen-Wilson, W.D. and L.K. Tyler. 'The temporal structure of spoken language understanding.' *Cognition* 8, 1980, 1-71.

Matthews, P.H. *Morphology.* Cambridge: C.U.P., 1974.

Metzing, D. *Frame Conceptions and Text Understanding.* Berlin: de Gruyter, 1979.

Miller, G.A. 'Informavores.' In: Machlup, F. and U. Mansfield. *The Study of Information: Interdisciplinary Messages.* New York: Wiley, 1984.

Miller, L.A., Heidorn, G.G., and K. Jensen. 'Text-critiquing with the Epistle System: an Author's Aid to Better Syntax.' *Proc. Nat. Comp. Conf.*, 1981.

Minsky, M.A. 'A Framework for Representing Knowledge.' In: P. Winston (Ed.) *The Psychology of Computer Vision* New York: McGraw-Hill, 1975.

Montague, R. 'The Proper Treatment of Quantification in Ordinary English.' In: Thomason, R.H. (Ed.) *Formal Philosophy. Selected Papers of Richard Montague.* New Haven: Yale University Press, 1974.

Morton, J. 'A functional model for memory.' In D.A. Norman (Ed.) *Models of Human Memory.* New York: Academic Press, 1970.

Morton, J. 'Word Recognition.' In: Morton, J. and J.C. Marshall (Eds.), *Psycholinguistics 2: Structures and Processes.* Cambridge, Mass.: MIT-Press, 1979a.

Morton, J. 'Facilitation in word recognition: experiments causing change in the Logogen model.' In: Kolers, P.A., M.E. Wrolstad, and H. Bouma (eds.), *Processing of Visible Language, Vol. 1.* New York: Plenum, 1979b.

Murrell, G.A. and J. Morton. 'Word Recognition and Morphemic Structure.' *Journal of experimental psychology* 102, 963-968, 1974.

Nes, F.L. Van. 'Analysis of keying errors.' *Ergonomics,* 19.2 (1976), 165-174.

Peer, W. van. 'Het anker leert nooit zwemmen, ook al ligt het steeds in het water.' *Moer,* 5, 1982, p.2-15.

Peterson, J.L. 'Computer programs for detecting and correcting spelling errors.' *Communications of the ACM,* 23(12) (1980a),676-687.

Peterson, J.L. *Computer Programs for Spelling Correction.* Lecture Notes in Computer Science. Vol.96. Springer Verlag, New York, 1980b.

Peterson, J.L. 'A Note on Undetected Typing Errors'. *Communications of the ACM,* 29(7) (1986),633-637.

Pijls F., W. Daelemans and G. Kempen. 'Artificial Intelligence Tools for Grammar and Spelling Instruction.' Submitted to Second International Conference on Children in the Information Age. Sofia, Bulgaria, 19-23 May, 1987.

Pounder, A. and M. Kommenda. 'Morphological analysis for a German text-to-speech system.' Proceedings of COLING, Bonn, 1986.

Prescott Loui, R. 'On Spelling Error Detection.'*Communications of the ACM,* 24(5), (1981), 331-332.

Quillian, M. R. 'Semantic Memory.' In: Minsky, M. (Ed.) *Semantic Information Processing.* MIT-Press, 1968.

Rumelhart, D.E. and J.C. McClelland. *Parallel Distributed Processing. Vol.I: Foundations.* Cambridge Mass.:MIT-Press, 1986a.

Rumelhart, D.E. and J.C. McClelland. 'On learning the past tenses of English verbs.' In Rumelhart and McClelland, 1986a, 1986b.

Santen, A. van. *Morfologie van het Nederlands.* Dordrecht: Foris Publications, 1984.

Schaerlaekens, A.M. *De Taalontwikkeling van het Kind.* Groningen: Wolters-Noordhoff, 1979.

Schane, Sanford A. *Generative Phonology* Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1973.

Schotel, H. and F. Pijls. 'Een prototype van grammaticaonderwijs op een Lisp machine.' *Informatie* 28 (12), 1985.

Schutter, G. de, 'Aspekten van de Nederlandse klankstruktuur'. *Antwerp papers in linguistics*, nr.15, 1978.

Selkirk, E.O., 'The Syllable' In: Hulst, H.v.d. and N. Smith (eds.), *The Structure of Phonological Representations*, Vol I and II, Dordrecht: Foris, 1982/83.

Shaffer, L.h. en J.Hardwick. 'Errors and error detection in typing.' *Q.Jl. of exp.Psych.*, 21 (1969), 209-213.

Simon, J.C. (Ed.) *Spoken Language Generation and Understanding.* Dordrecht: Reidel, 1980.

Sleeman D. and Brown J. (eds) *Intelligent Tutoring Systems.* New York: Academic Press, 1982.

Smedt, K. de. 'Orbit, an Object-Oriented Extension of Lisp.' Internal Report 84FU13, K.U. Nijmegen, Psychological Laboratory, 1984.

Smedt, K. de. 'Using Object-Oriented Programming Techniques in Morphology and Syntax Programming.' In: O'Shea, T. (Ed.) *ECAI 84 Proceedings*, 1984.

Smedt, K. de. 'Object-Oriented Programming in Flavors and CommonOrbit.' In: Hawley, R. (Ed.) *Artificial Intelligence Programming Environments.* Ellis Horwood, 1987 (forthcoming).

Smedt, K. de and G. Kempen. 'Incremental sentence generation'. In: G. Kempen (Ed.) *Natural Language Generation: Advances in Artificial Intelligence, Psychology and Linguistics.* Dordrecht: Kluwer Academic Publishers (in preparation).

Smith, R.N. 'Computer Aids to Writing.' In: W. Frawley (ed.), *Linguistics and Literacy.* New York: Plenum, 1982, p. 189-208.

Srihari, S.N., J.J. Hull and R. Choudhari. 'Integrating Diverse Knowledge Sources in Text Recognition.' *ACM Transactions on Office Information Systems*, 1(1), 1983, 68-87.

Steels, L. *Representing Knowledge as a Society of Communicating Experts.* MIT: AI Lab Treatise, TR-542, 1980.

Steels, L. 'Programming with Objects.' Schlumberger Doll Research, AI-Memo 9, 1981a.

Steels. L. 'Programming with Objects Using Orbit.' Schlumberger Doll Research, AI-Memo 13, 1981b.

Steels, L. 'An Applicative View of Object-Oriented Programming.' Schlumberger Doll Research, AI-Memo 15, 1982.

Steels, L. 'Community Memories'. AI-Memo 3/M2, 1985.

Steels, L. 'The KRS Concept System.' Vrije Universiteit Brussel. Artificial Intelligence Laboratory. Technical Report 85-4. Brussels, Belgium, 1985.

Steels, L. 'Second Generation expert Systems.' In: *Future Generation Expert Systems.* Amsterdam: North-Holland, 1985.

Steels, L. and K. De Smedt. 'Some Examples of Frame-based Syntactic Processing.' In: F. Daems and L. Goossens (eds.) *Een Spyeghel voor G. Jo Steenbergen.* Leuven: Acco, 1983, 293-305.

Steels, L. and W. Van de Velde. 'Learning in Second Generation Expert Systems.' In: Kowalik (Ed.) *Knowledge Based Problem Solving.* New Jersey: Prentice Hall, 1985.

Stefik, M. and D.G. Bobrow. 'Object-Oriented Programming: Themes and Variations.' *AI-Magazine.* 6(4), 1986, 40-62.

Taft, M. 'Prefix stripping revisited.' *Journal of verbal learning and verbal behavior* 20, 1981, 289-297.

Taft, M. and K.I. Forster. 'Lexical storage and retrieval of prefixed words.' *Journal of verbal learning and verbal behavior* 14, 1975, 638-647.

Taft, M. and G. Hambly. 'The influence of orthography on phonological representations in the lexicon.' *Journal of memory and language* 24, 1985, 320-335.

Teitelman,W. *Interlisp Reference Manual.* Xerox Research Center, Palo Alto, Calif. (1978) Ch.17.

Toorn, M.C. van den. 'De tussenklank in samenstellingen waarvan het eerste lid een afleiding is.' *Nieuwe Taalgids* 74, 1981a, p.197-205.

Toorn, M.C. van den. 'De tussenklank in samenstellingen waarvan het eerste lid systematisch uitheems is.' *Nieuwe Taalgids* 74, 1981b, p.547-552.

Toorn, M.C. van den. 'Tendenzen bij de beregeling van de verbindingsklank in samenstellingen I.' *Nieuwe Taalgids* 75, 1982a, p.24-33.

Toorn, M.C. van den. 'Tendenzen bij de beregeling van de verbindingsklank in samenstellingen II.' *Nieuwe Taalgids* 75, 1982b, p.153-160.

Trommelen, M., *The Syllable in Dutch.* Dordrecht: Foris, 1983.

Uit den Boogaart, P.C. (ed.), *Woordfrequenties in geschreven en gesproken Nederlands.* Utrecht: Oosthoek, Scheltema en Holkema, 1975.

Weinreb, D. and D. Moon *Lisp Machine Manual.* Symbolics Inc., 1981.

Wester, J. 'Language Technology as Linguistics: A phonological case study of Dutch spelling.' In: Bennis, H. and F. Beukema (Eds.) *Linguistics in the Netherlands 1985.* Dordrecht: Foris, 1985a.

Wester, J. 'Autonome spelling en toegepaste fonologie; of: naar een generatieve spellingtheorie.' *Gramma* 9(3), 1985b, 173-196.

Wijk, C. van and G. Kempen, 'From sentence structure to intonation contour'. In: B. Muller (Ed.), *Sprachsynthese.* Hidesheim: Georg Olms Verlag, 1985, p. 157-182.

Winograd, T. 'Frame Representations and the Declarative/Procedural Controversy.' In: Bobrow and Collins (Eds.) *Representation and Understanding: Studies in Cognitive Science* New York: Academic Press, 1975, p. 185-210.

Winograd, T. *Language as a Cognitive Process. Volume I: Syntax.* Reading Mass.: Addison-Wesley, 1983.

Winston, P.H. *Artificial Intelligence.* Reading Mass.: Addison-Wesley, 1984.

Woordenlijst van de Nederlandse Taal. 's Gravenhage: Martinus Nijhoff, 1954.

Yannakoudakis, E.J. and D. Fawthrop. 'The rules of spelling errors.' *Information Processing and Management,* 19.2 (1983a), 87-99.

Yannakoudakis, E.J. and D. Fawthrop. 'An intelligent spelling error corrector.' *Information Processing and Management,* 19.2 (1983b), 101-108.

Yazdani, M. 'Intelligent Tutoring Systems Survey' *Artificial Intelligence Review* 1, 1986, 43-52.

Yianilos, Peter N. 'A dedicated comparator matches symbol strings fast and intelligently.' *Electronics* (december 1983), 113-117.

Zamora, A. 'Automatic detection and correction of spelling errors in a large database.' *J.Am.Soc.Inform.Sci.,* 31(1) (1980), 51-57.

Zamora, E.M., J.J.Pollock and A.Zamora. 'The use of trigram analysis for spelling error detection.' *Information Processing and Management,* 17(6) (1981), 305-316.

Zoeppritz, M. 'Human Factors of a "Natural Language" Enduser System.' In: Blaser, A. and M. Zoeppritz (eds.) *Enduser Systems and Their Human Factors.* Berlin: Springer-Verlag, 1983.

Zonneveld, W. 'Autonome Spelling.' In *De Nieuwe Taalgids* 73 (6), 1981.