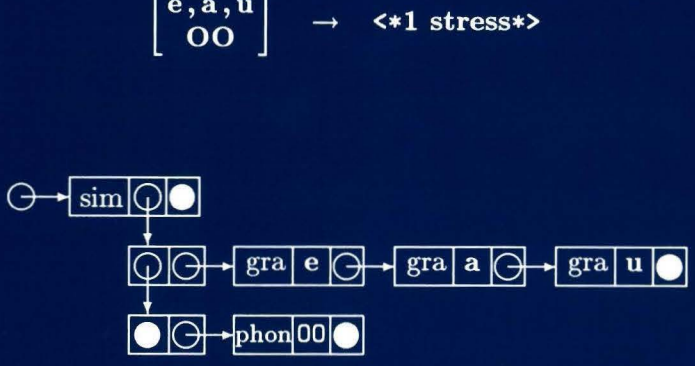
TooLiP : a development tool for linguistic rules

TooLiP:
A Development Tool for Linguistic Rules

Hugo van Leeuwen



Cover: Three aspects which are typical for TooɟiP are illustrated on the cover. The topmost figure is a linguistic rule, wich assigns primary word stress to vowels which are pronounced as /o/ and written as 'eau'. The middle figure illustrates the internal representation of the focus of this rule. The bottom figure illustrates the internal data structure of synchronized buffers, and how, moving from left to right through the grapheme buffer, one can access the phoneme buffer.

TooLiP:
A Development Tool for Linguistic Rules

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus, prof. ir. M. Tels,
voor een commissie aangewezen door het College van Dekanen
in het openbaar te verdedigen
op vrijdag 15 december 1989 te 14.00 uur

door

Hugo Cornelis van Leeuwen

geboren te Castricum

Dit proefschrift is goedgekeurd door de promotoren:

Prof. dr. S.G. Nooteboom
Prof. dr. H. Bouma

The work described in this thesis has been carried out at the Philips Research Laboratories as part of the Philips Research programme.

Acknowledgements

WRITING a thesis is generally hard and solitary labour. I am no exception to this rule. Nevertheless, I wish to emphasize that neither form nor content of this thesis would have had the same quality, had the labour been purely solitary.

I definitely consider the scientific climate at the Institute for Perception Research (IPO) an important factor in the accomplishment of this thesis. Although the contribution to this thesis of any individual person can hardly be measured or traced, the numerous scientific and social discussions which constitute the pleasant working atmosphere at IPO, have certainly been very stimulating.

There are a few persons whom I wish to thank personally. This concerns first of all my promotor Sieb Nooteboom. He has been the main stimulating force all along the way. I have allways found his comments and suggestions remarkably to the point, and I feel they have very much improved the manuscript.

Further I wish to thank Marc van Leeuwen and Kees van Deemter for their substantial help in the effort of formalizing the notion of complementation as described in chapter 3. I thank John de Vet for his scrupulous reading of chapter 4, to a level at which he was able to suggest some improvements to the algorithms.

As to the form of this thesis, I wish to thank Marc and David van Leeuwen for their TEXnical assistance. Without their TEXpertise and readiness to answer and solve my numerous questions, this thesis would not have the TEXnical quality in which I now take pride.

Contents

List of Figures

List of Tables

List of operational definitions

THE following list gives a description of the most important concepts which are introduced in this thesis, and of some general concepts which are used in a specific manner here. All terms are introduced in the course of the study, but the reader may wish to consult the list at another moment. More information about the concepts can be found on the pages listed behind the item.

In this list a description of the terms has been given rather than a formal definition. The definition or the main source of information of a concept can be found at the underlined pagenumbers. Examples of (the use of) the concept can be found on the *italicized* pages.

Action: the application of a rule; the structural change of a linguistic rule is added to the output and synchronized with the input segments which are associated with the focus of the rule; 19.

Alternative operator: the 'or' operator for patterns; 17, 46, 64, 88, *93*, 98.

Candidate: a string is a candidate if it can be fitted to a pattern such that its segments do not match the complemented part, but match the non-complemented parts of the pattern; 55.

Common rule: a linguistic rule which can be triggered by more than one segment. Operates further like a segment rule; *20*, 22, *143*.

Complementation operator: the 'not' operator for patterns; 18, 48, *54*, 64, 92, 100, 150.

Compositionality: a formalism is compositional if the meaning of an arbitrary expression can be expressed as a function of the meaning of the composing sub-expressions; 47, 55, 70.

Concatenation operator: the operator which concatenates patterns. The specified patterns must be found in succession; 17, 47, 64, 86.

Notational Conventions

THROUGHOUT this thesis it is attempted to maintain consistency in terminology and notation. In addition to this overview, in each chapter the relevant conventions and terminology are explained. Generally this is consistent between the chapters, but in one case it is not, as explained below.

Generally, when basic notions are introduced they are printed *slanted*. Data such as linguistic rules, patterns and buffer contents, which are present or may be present in a computer program, are printed in `typewriter style`. However, as explained below, linguistic rules and patterns are also noted in paper and pencil notation, in which case the are printed in **bold face**. Algorithms are typeset in the following manner: keywords are printed in **bold face**, procedures and functions have their first letter capitalized and are printed in *Italics*, and variables and types are printed with no capitals and also in *italics*. Finally, the names of actually developed linguistic modules are printed in SMALL CAPS.

The way in which linguistic rules and patterns are typeset depends on the angle of approach. In chapter 2 the system is approached from the user's point of view. Therefore, all examples of linguistic rules in this chapter are displayed in the exact appearance they have in the user-created computer files. The other chapters take somewhat more distance. In these chapters the rules are displayed in a paper-and-pencil notation. The 'and' insertion rule, for instance, which is discussed in 5.2.1, would be displayed as in (1) in chapter 2, and as in (2) in the other chapters.

```
t -> &,t / [ D ] _ [  D ]                    (1)
           ['0 ]   ['{0}]
                     {1}
```

$$\mathtt{t} \rightarrow \mathtt{\&,t} \;/\; \begin{bmatrix} \mathbf{D} \\ \neg\mathbf{0} \end{bmatrix} \_ \begin{bmatrix} \mathbf{D} \\ \neg\left\{ \begin{matrix} \mathbf{0} \\ \mathbf{1} \end{matrix} \right\} \end{bmatrix} \qquad (2)$$

The braces, which can span several lines in the paper and pencil notation, are split up in the computer implementation, where they are repeated on each line to indicate the arguments. The complementation sign '¬' gets an ASCII equivalent: '~'.

Finally, throughout the thesis, phonemes are used in examples. Table I gives the relation of the coded ASCII representation used in TooIiP to the IPA (International Phonetic Alphabet) representation.

Table I: Conversion table of phonemes. For each phoneme the IPA representation, the TooIiP representation and a Dutch example word are listed.

IPA	TooIiP	example	IPA	TooIiP	example
u	U	r*oe*t	p	P	*p*as
ui	UJ	r*oei*t	pj	PJ	boom*pj*e
o	OO	r*oo*d	t	T	*t*as
oi	OJ	h*ooi*t	tj	TJ	*tj*alk
ɔ	O	r*o*t	k	K	*k*an
ɔi	OI	h*oi*	f	F	*f*ok
ɑ	A	m*a*t	s	S	*s*ok
ɑi	AI	det*ai*l	ʃ	SJ	*ch*auffeur
ɑu	AU	k*ou*d	χ	X	*g*ok
a	AA	m*aa*t	b	B	*b*as
ai	AJ	m*aai*t	d	D	*d*as
ɛ	E	l*e*s	dj	DJ	*dj*atiehout
ɛi	EI	r*ei*s	g	G	*g*oal
ɪ	I	p*i*t	v	V	*v*uur
e	EE	l*ee*s	z	Z	*z*eer
eu	EW	l*eeu*w	ʒ	ZJ	*j*ournaal
i	II	l*ie*p	m	M	*m*eer
iu	IW	k*ieu*w	n	N	*n*eer
y	Y	m*uu*r	ɲ	NJ	ora*nj*e
ʌy	UI	m*ui*s	ŋ	Q	ba*ng*
ø	EU	k*eu*s	l	L	*l*ang
œ	OE	p*u*t	ɬ	LL	apri*l*
ə	C	d*e*	r	R	*r*ok
ʔ	GS	*glottal stop*	w	W	*w*eer
	SI	*silence*	j	J	*j*an
			h	H	*h*ok

Chapter 1

Introduction

IN reading aloud text one converts strings of letters into sounds. Although for many of us this may seem a fairly simple feat, the processes involved are not as simple as it may seem. This is soon found out if we try to automatize letter-to-sound conversion, for example in a machine or computer program for converting text to speech.

Part of the complexity stems from the fact that there is no one-to-one correspondence between the letters as they are used in the orthography of the language and the sound of speech. In reading a word like 'development', for example, we encounter three times the letter 'e'. In the orthography they are indistinguishable, but in the spoken version they are all different, the first one sounding as in 'beach', the second one as in 'help' and the third one as in 'the'. Also, the second 'e' bears word stress and for quick understanding by a listener we should get all of these factors right.

Now suppose we were to devise a reading machine, that is, a machine that converts automatically given text into intelligible speech, then this machine is confronted with the same problem. Of course, we can explicitly tell the machine how to pronounce it, just like our parents told us how to pronounce many words. However, they have never given us the pronunciation of all words in our native language. Once we acquire some feeling for language, and this can be quite early, we are capable of figuring some of it out ourselves. So apparently we acquire rules on how to pronounce words. Some of them are explicit, learnt at school, but most of them will probably be implicit.

Finding these rules and formulating them explicitly is best done by trained linguists. Such rules can then be used in the reading machine. Of course, such rules will probably not cover the whole of the language, since many languages have numerous exceptions to their regularities, but generally the regularities and sub-regularities, which can be expressed elegantly in rules, cover a respectable part of a language.

Another observation also motivates the development of a rule component in the pronunciation module of the reading machine. Suppose we were to

pursue the strategy of coaching, and we were to store all words of a language in a lexicon rendering pronunciation and word stress. We would, first of all, have a problem with storing *all* words. Constantly new words arise and old words disappear. The new words, often denoting a new phenomenon, typically appear quite suddenly and with relatively high frequency. It would therefore be annoying if the reading machine, used to read aloud a newspaper text, would fail on those words. A second problem would be the law of diminishing returns. While the 200 most frequent words in English cover over fifty percent (53.6 %) of the words in running text ("Brown corpus", Kucera & Francis, 1967), the next 800 words only increase the score 15.3 %, a trend which is only persevered more strongly for words of lower frequency. Thus, although a small lexicon is remarkably productive, increasing its size will yield quickly diminishing returns. A rule component therefore serves both completeness and efficiency.

Apparently we need a rule component in the reading machine. The task is now to find the relevant rules. When this is done with paper and pencil we find that, while one rule may be very clear and simple, a whole set of simple rules can form a complicated prescription whose correctness or desired functionality is difficult to establish. Here computers may provide help, as they are very good in performing a sequence of simple instructions. If we were to devise a tool which could read the paper and pencil rules, we could have the machine evaluate the rules quickly on all kinds of text input, and thus we could concentrate on the functionality of the rule set rather than having to put effort each time into the deterministic process of evaluating our rules.

In several places this approach has indeed been followed (Carlson & Granström, 1976; Elovitz, Johnson, McHugh & Shore, 1976; Hertz, 1981; Hertz, Kadin & Karplus, 1985; Holtse & Olsen, 1985; Karttunen, Koskenniemi & Kaplan, 1987; Kerkhoff, Wester & Boves, 1984; Kommenda, 1985). The characteristics of these systems, which are studied more closely in one of the following chapters, differ between the systems, but they all have in common that linguistic rules are expressed in some format and executed by machine. For a variety of languages the suitability of rules for expressing spelling to pronunciation has been established.

This thesis describes yet another tool for the development of linguistic rules. Like all other systems it has been designed with special intentions and for particular purposes for which existing systems appeared not to be suited or simply were not available. The main motivation for designing our own system is that, apart from being used in the reading machine, it is also intended to be used as an analysis tool to collect statistical information on spelling to pronunciation relationships—for instance, how often is an 'e' pronounced

as in 'beach', compared to the realizations as in 'help' and 'the'—which is another question of interest in the field of linguistics. When the conversion is performed by rules, the information is practically free: the rules explicitly note the relationship, only some additional effort of an administrative nature is needed.

Apart from this specific requirement, the system should meet some other, rather general requirements. First of all linguists should be able to address it in a familiar manner—the system should accept the rules which are formulated as closely to the paper and pencil notation as possible in a computer implementation. The possibilities for expression should be as little restricted as possible. Next, the system should feature tools to facilitate the development of a rule set. Apart from diagnostic messages when the syntax is violated, it should be able to provide detailed but carefully dosed information on the derivational process for debugging purposes. Further, apart from being able to access the input-to-output relationship from the outside, i.e., having these relationships available when a word has been converted, they must also be available inside, during the conversion process, so for instance one must be able to test if a particular pronunciation is derived from a particular character sequence. Finally, from the engineering point of view it is desirable to separate linguistic knowledge from the execution machine. The more linguistic knowledge is declared explicitly, for instance what are the pronunciation codes, which of these are defined as consonantal, etc., the less attached to a particular linguistic flavour the system will be and thus the more independent of the application. The only thing the system should offer is a certain inference mechanism and an environment to provide this inference mechanism with meaningful rules.

In this thesis, a tool for the development of linguistic rules is described which satisfies the above requirements. It is called TooɫiP, which stands for "Tool for Linguistic Processing" and is pronounced in the same way as the Americans pronounce 'tulip'. The link to this typical Dutch flower seems appropriate since the system both originated in Holland and has been used to develop rules for the spelling-to-pronunciation conversion (also called grapheme-to-phoneme conversion) of Dutch.

Many of the examples will be taken from this application. The system is not, however, restricted solely to this application, nor to the Dutch language. The general characterization is that it is a tool with which one can test and implement phonological theories. It is a tool with which one can define almost any transcription of input characters to output characters which can be guided by rules. For instance, it has also been used to spell out integer numbers and acronyms, and can probably also be applied advantageously to spell out abbreviations or correct root mutation due to morphological processes. In

fact, probably any rule-based segmental conversion or transcription process can be implemented in TooIJP, which makes the system suited to be used in quite a variety of modules of the reading machine.

In this thesis TooIJP will be treated from several points of view. In chapter 2 a user's point of view is taken, and TooIJP is described as it presents itself to the user[1]. First the possibilities available for constructing linguistic rules are described, and and an explanation is given of the type of manipulations one can express with the rules. Next it is explained how one can group these rules into a set which defines a conversion scheme, i.e., a prescription of how to compute the output from a given input. Then the development support which the system provides is discussed and a short comparison with some other systems is made on the basis of the properties discussed in this chapter.

In chapter 3 a mathematician's point of view is taken. It concerns a specific aspect of TooIJP, which remains underexposed in chapter 2. In the linguistic rules the user specifies target and context patterns which generally denote a set of strings. For this purpose an extended form of regular expressions (a widely used mathematical tool) is used. The extension consists of adding some operators, i.e., mechanisms to express certain relationships between regular expressions, to the formalism. The introduction of one operator, the complementation operator, specifically gives rise to an unexpected problem. The complementation operator is used to express the desired absence of a pattern, and is desirable for elegant pattern description. In the intention of restricting the user as little as possible, a full, unrestricted availability of this operator is pursued. The problem which arises is that introduction in a straightforward manner, viz. defining complementation analogously to how it is defined in set theory, leads to an unexpected interpretation for a certain class of patterns. That is, the formal interpretation differs from the subjectively expected meaning. This is considered to be undesirable, and therefore an alternative formal interpretation is proposed in chapter 3.

In chapter 4 a technical point of view is taken, concerning the implementation of the system. Three important aspects are discussed in detail. The first one concerns the internal representation of the user-specified patterns. The second concerns the matching strategy, i.e., how patterns are evaluated. A full algorithmic description is given. The third concerns the system's internal data structure which is used to support the requirement of providing input-to-output relations.

[1]Chapter 2 is a slightly modified form of a previously published article: Van Leeuwen, H.C. (1989); A development tool for linguistic rules, *Computer, Speech and Language*, **3**, 83–104. Compared with the article, the exposition of the linguistic component (section 2.3) has been altered, and Appendix 2.A which contains a formal specification of this linguistic component has been added.

In the last chapter, chapter 5, the merits of TooIjP are considered. First some applications for which it has been used are discussed. As an example of how TooIjP can be used it is discussed in detail how the spelling out of integer numbers can be achieved. From somewhat more distance the major application is viewed, viz. the grapheme-to-phoneme conversion. Next, the complementation operator is reviewed and the formalism proposed in chapter 3 is evaluated. Then, TooIjP is compared with seven existing systems designed for similar purposes. This comparison is more extensive and complete than the one in chapter 2, since here all the properties discussed in the previous chapters are included. The conclusions of these sections, how TooIjP is used in practice and how it relates to existing systems, lead to the proposal of five possible extensions of the system, and these conclude the thesis.

Chapter 2

A development tool for linguistic rules[1]

Abstract

In this chapter the TooIJiP system is presented. It is a development tool for linguistic rules, and with it one can develop and test a set of linguistic rules which define a scheme to convert an input string to an output string. The system is approached from the point of view of linguists, since they are the main users of such a system.

First the basic configuration is discussed. Linguistic rules are the user's main tool to manipulate input characters. The possibilities for transcribing input characters and the facilities to test contexts are described. Grouping these rules into a module provides a mechanism to manipulate strings. Modules are concatenated in a conversion scheme to perform their tasks in the desired order. The system can provide feedback on the conversion process, both for purposes of debugging and efficiency improvement.

A special characteristic of TooIJiP is that input-to-output relations are preserved. On the one hand this means that one can make use of derivational information in the linguistic rules, and on the other that the system can be used to gather statistics on input-to-output relations. Given the major application for which TooIJiP has been used, viz. a grapheme-to-phoneme conversion system, TooIJiP can be used as an analysis tool for statistics on grapheme-to-phoneme relations.

Finally, some extensions are discussed which are included to increase its user-friendliness and applicability. Also, some characteristics of the system are discussed and compared to those of some other systems. A short survey of the applications in which it is used concludes the chapter.

[1]This chapter is a slightly modified version of a previously published article: Van Leeuwen, H.C. (1989); A development tool for linguistic rules. *Computer, Speech and Language*, **3**, 83–104.

2.1 Introduction

SINCE the publication of the Sound Pattern of English (SPE) by Chomsky & Halle (1968), linguistic re-write rules have become very popular in phonology. This is due to the fact that rules of this type have proved to be an elegant and efficient tool for formulating phonological processes. With the rise of language- and speech technology such rules also found a wider application, as they appeared to be adequate for the symbol manipulation which is needed there.

One specific area is the development of text-to-speech systems. In most Indo-european languages the spelling is not phonetic, i.e., the correspondence between spelling and pronunciation is not one-to-one. Therefore, generally a conversion phase is needed to assign a sound representation (phonemes) to the orthographic text (graphemes). This phase is called grapheme-to-phoneme conversion.

For a variety of languages the usefulness of linguistic re-write rules for grapheme-to-phoneme conversion has been investigated (Ainsworth, 1973; Carlson & Granström, 1976; Elovitz, Johnson, McHugh & Shore, 1976; Hertz, 1981; Holtse & Olsen, 1985; Kerkhoff, Wester & Boves, 1984; Kommenda, 1985; Kulas & Rühl, 1985; Van Leeuwen, Berendsen & Langeweg, 1986). It is widely agreed that these rules serve well for the large majority of regularities in pronunciation of most languages, but that they should not be considered as the best tool for irregularities (e.g., 'though' ↔ 'through') or ambiguities (e.g., 'I read' (present tense) ↔ 'I read' (past tense)). For irregularities, alternative approaches are more adequate, such as morph-based or word-based lexica, where phonetic transcription is stored in a database as a function of the orthography. For ambiguities higher level linguistic processing seems necessary, such as syntactical- and word class analysis. Therefore, in realistic applications a combination of approaches is often encountered (Allen, Hunnicutt & Klatt, 1987).

In this chapter the TooIjP system is described, the main purpose of which is to enable a linguist to develop an ordered set of linguistic re-write rules, which defines a scheme to convert an input string into an output string. The user can choose whether the input and output string are of the same type or not. In the first case, a one-level concept is used, characters form the input and characters result. Used in a grapheme-to-phoneme context, the interpretation of graphemes and phonemes is done at the cognition level of the (linguist) user. In the other case a two-level concept is used, for instance one level corresponds to grapheme input and the other to phoneme output. A special feature of TooIjP is that, once the second level has been initiated, the alignment between the levels—i.e., for instance the correspondence between

graphemes and phonemes—can be addressed in the rules. So, in the two-level concept the notion of grapheme and phoneme can be entered into the system and used explicitly in the rules.

The presence of co-ordinated information on orthography and pronunciation is—for the purpose of grapheme-to-phoneme conversion—not a necessity from a theoretical linguistic point of view, but can be convenient for a number of applications. For instance, stress assignment in Dutch can profit from this information, the rules can be formulated elegantly, and be well separated from other parts of the grapheme-to-phoneme conversion. Also, the relation between graphemes and phonemes is an attractive by-product of the conversion. For applications where this information is needed (see for instance Lawrence & Kaye, 1986), the grapheme-to-phoneme rules are sufficient and no alignment algorithm is needed. Also, statistical information on individual grapheme-to-phoneme relations can be gathered very easily. By running a sample text corpus through the rules, it can be established how many different phonemic realizations a specific grapheme sequence has, and what the frequency of occurrence is for each realization.

For some applications it is perhaps a limitation of TooɹjP that only one output string results from an input string. The system is not designed to generate all possible outputs for input which has different possible correct transcriptions. For instance, in a word like 'either' the first vowel may be pronounced both as an /i:/ or as an /ai/, but in TooɹjP one must choose one or the other. Also, a word like 'object' is ambiguous if the word class is unknown ('objèct' (verb) ↔ 'òbject' (noun)), and without this information the correct pronunciation cannot be established. However, if such disambiguating information is present (for instance provided by a separate process) TooɹjP *is* able to process this kind of non-segmental information and determine the correct pronunciation (given the appropriate rules).

It was decided not to implement a facility which could produce all possible correct transcriptions for ambiguous input. As to the first type of ambiguity, when both transcriptions are correct and interchangeable, producing only one of the alternatives is not erroneous. As to the second type, where the ambiguity can be resolved by additional, non-segmental information, it is preferable to aim at the correct transcription by providing the system with that information. Since—for the purpose of text-to-speech conversion—the processes involved are to a large extent deterministic and unambiguous, it was felt that the additional possibilities would not justify the cost of increased complexity.

On the other hand, an output string always results, no matter how incomplete or incorrect the specified rules may be. The output string may not be the desired one, but it will never cause TooɹjP to crash, which is important

when it is used, for instance, as part of a text-to-speech system.

TooljP has been conceived in the first place to serve as a development tool for linguistic rules which define a grapheme-to-phoneme conversion (Berendsen, Langeweg & Van Leeuwen, 1986; Berendsen & Don, 1987). Therefore, some choices in the design have been tuned to this application. I believe that the system can potentially be applied in a wider area, viz. in all cases where re-write rules are used to express certain linguistic processes. The discussion of TooljP, however, will mainly be done from the viewpoint of grapheme-to-phoneme conversion, since most of the experience with the system has been gathered in this application.

This chapter has the following structure. First, the facilities which are needed to advantageously specify a conversion scheme are discussed. Then the facilities offered by TooljP are described: the basic units on which linguistic rules operate, the different types of rules, how rules must be combined in a module and how the modules constitute a conversion scheme. A description is then given of the information the system contains once a conversion scheme has been developed, and of the means which are available to the user to extract this information. This is TooljP's basic configuration. Next some extensions are discussed which have been included to improve the flexibility and user-friendliness. Finally, some characteristics of TooljP are compared with those of some other systems, and the applications in which it is used are discussed.

2.2 Linguistic needs

The basic units the linguist user wants to manipulate are the input characters. Via a certain scheme of transcriptions the input is manipulated into the desired output, for instance, the phonemic transcription of the input string. In TooljP the input and output characters are called *segments*. The segments are user-defined, and in the application of grapheme-to-phoneme conversion the input segments are called *graphemes* and the output segments *phonemes*[2].

The basic mechanism with which one can manipulate these segments is the *linguistic rule*. Often the phonemic transcription of a grapheme depends on the context: a 'c' sounds different before an 'a' than before an 'e'. In the field of linguistics a particular type of re-write rules introduced by Chomsky & Halle (1968), has become very popular for this purpose. The general format of these rules is as follows:

```
F -> C / L _ R                                    (2.1)
```

[2]The relation between these and most other basic concepts, which are printed slanted when introduced, is given in Fig. 2.1.

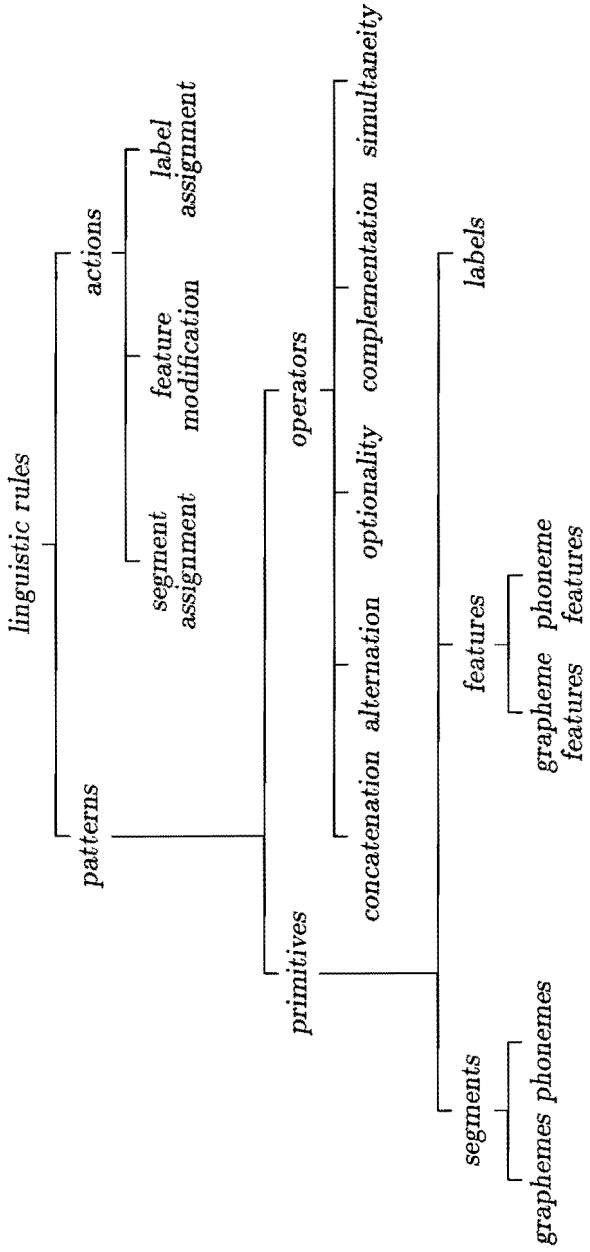


Figure 2.1: Relation between basic concepts. See text for explanation.

> A certain focus 'F' in the input string is re-written into the structural change 'C' if the focus is preceded by a left context 'L' and followed by a right context 'R'.

There are different types of linguistic processes the user may want to account for, which the rules must thus be able to express. First of all, transcription rules are needed, which assign phonemes to graphemes or vice versa. Then, one may want to insert segments or boundaries (affix-stripping; sadly → sad+ly), modify segments (root mutation; happi+ly → happy+ly) or delete symbols (the final 'r' in British English, which is not pronounced). Also, one may wish to express phonological generalizations, such as the devoicing of final obstruents in Dutch and German, in one general rule. This is usually called feature modification. Finally, one may want to represent and use non-segmental information, such as stress-level or word class, and be able to manipulate this kind of information.

One linguistic rule will seldom express the whole of the transcription one wants to perform. Therefore, one may want to group a set of rules which together perform a certain task, and separate it from another set, which performs a different task. For instance, it seems desirable to insert affix- and morph boundaries into the orthographic input before actual phoneme assignment is done. This is called modularization.

These two mechanisms, the linguistic rule and grouping the rules into a module, suffice for most of the transcription tasks the user wants to specify. They will therefore be described in the following section, which describes the main body of TooljP. Extensions to this configuration will be discussed in subsequent sections.

2.3 The linguistic component

The basic architecture of TooljP is depicted in Fig. 2.2. It consists of three layers: the linguistic rule, the *module* and the *conversion scheme*. The linguistic rules operate on specific segments: the input segments denoted by the focus are transcribed into the output segments which are denoted by the structural change. The linguistic rules can be grouped into a module. A module operates on a string: the string in the module's input buffer is converted—in accordance with the specification of the linguistic rules in the module—to an output string which is written to the output buffer. Finally, the modules are grouped into a conversion scheme, which is defined by consecutive application of the modules.

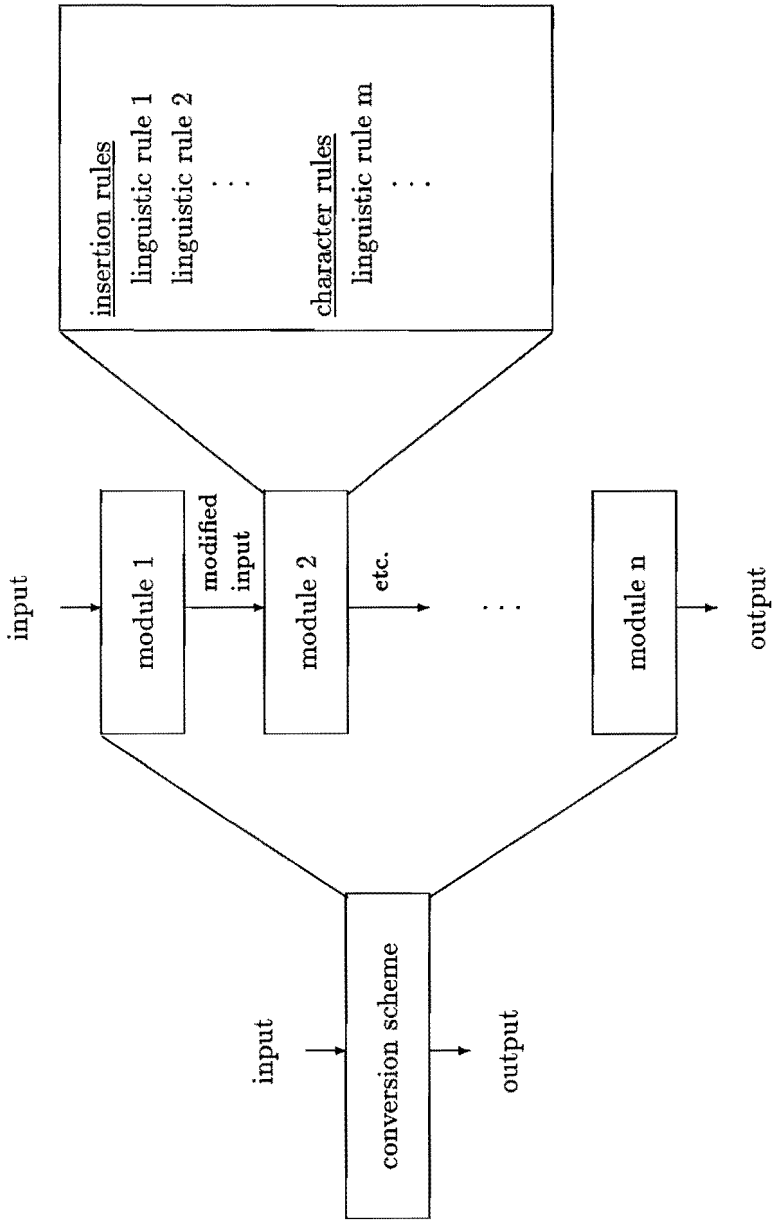


Figure 2.2: TooLiP's architecture.

In this section, TooIJiP's architecture will be discussed from small to large[3]. First the linguistic rule will be considered, then it is discussed how rules are organized in a module, and finally it is explained how a conversion scheme is built.

2.3.1 Linguistic rules

In (2.2) the general format of a linguistic rule is given once again.

```
F -> C / L _ R                                    (2.2)
```

The focus (F) of a linguistic rule refers to a sequence of zero or more segments in the input. The left (L) and right (R) contexts can also refer to the output segments. In general, F, L and R refer to a set of segment sequences which are called patterns. The structural change (C) is a sequence of zero or more segments which are added to the output and aligned with the focus in the input, if the rule applies.

TooIJiP adds segments to the output and aligns them with the input, rather than transcribing and substituting segments in the input. This is necessary to keep track of the derivation and to be able to refer to both input and output.

An example of a linguistic rule is given in (2.3), which serves to provide the pronunciation of the 'ch' in the French word 'cachet' /kaʃɛ/ (cachet)[4].

```
c,h -> SJ / <+segm,-cons> _ e,t          ! cachet    (2.3)
```

A 'ch' is pronounced as a 'SJ' /ʃ/, when it is preceded by a vowel and followed by the sequence 'et'[5]. The exclamation mark is a comment marker, so that one can comment on the purpose of the rule; all text behind this mark is ignored.

The different aspects of the linguistic rule will now be discussed in order. First the basic building block of a linguistic rule, the *primitive*, is dealt with. When a primitive is used in a linguistic rule it refers to exactly one segment in the input or output. Next, *patterns* are discussed. Patterns generally denote a set of strings, one of which must be present in the input or output. Patterns are constructed of primitives and operators; the primitives refer to the segments in the input and output, the operators specify how they are

[3] A formal specification is given in Appendix 2.A.

[4] Like all other examples in this chapter, (2.3) is displayed exactly the way the linguist types them in an input file.

[5] A conversion table of phoneme symbols to IPA symbols is included in Table I (page xv of the preface).

to be combined. Finally, the *actions* are dealt with. When a rule matches, i.e., when the patterns of focus, left and right context match, the structural change is added to the output and aligned with the input. This is called an action. These three notions, the building blocks, the patterns and the actions will now be discussed in order.

Primitives

Primitives are the building blocks of the linguistic rules. As mentioned, a primitive (in the rules) refers to one segment (in the input or output). To be precise: the primitive states the restrictions for a specific segment which must be met in order to have the pattern match. For instance, the '`e`' in (2.3) refers to the first character to the right of the focus. It matches if indeed an 'e' is found. But whether or not an 'e' is actually present in the input does not affect the fact that the primitive '`e`' refers to the first character to the right of the focus. In the same way '`t`' in (2.3) refers to the second character to the right of the focus.

Now '`e`' and '`t`' are primitives which are rather restrictive: only 'e' and 't' as segments in the input meet these restrictions, respectively. In (2.3) also another, less restrictive primitive is specified: '`<+segm,-cons>`'. All graphemes which are segmental but are not consonants match this primitive. This is the set of vowels, so 'a', 'e', 'i', 'o', and 'u' match. In general there are three different kinds of primitives: *segments*, *features* and *labels*. These will be discussed in order.

Segments. Segments in the linguistic rules have a one-to-one correspondence to the segments in the input or output, and are represented identically, i.e., an '`a`' in the rules expresses the desired presence of an 'a' in the input. Although segments in the rules and segments in the input or output are not exactly the same notion, they are called the same since the difference is so small, and generally it will be clear which is meant.

Segments are user-defined. They are coded by one ore more (ASCII) characters. In this way a linguist is not forced into a certain notational framework. He can decide himself how many phonemes he needs and whether there needs to be a distinction between allophonic variants or not. In this thesis graphemes are coded with one lowercase character, and phonemes with one or more uppercase characters. This would appear to exclude capital letters as graphemes, but there is a way around this, which will be explained when TooLiP's architecture is treated in more detail.

Features. The notion of binary features is well known from linguistics, where they describe phonological properties of phonemes. With these features strong

descriptive rules can be formulated. Since different graphemes can be thought of as sharing certain properties, just as phonemes do, the user can define features for both graphemes and phonemes. He can determine the number of features he needs and their symbolic representation. Every feature must receive a binary value, '+' or '-', for each appropriate segment.

In correspondence with graphemes and phonemes, the grapheme features are denoted in lowercase characters, and the phoneme features in uppercase characters. In the linguistic rules features are enclosed in angled brackets. For instance, the phoneme feature 'sonorant' is denoted as '`<+SON>`', and vowels on the grapheme level can be referred to as '`<-cons,+segm>`'.

Behind a feature specification an *identity marker* may be placed. Identity markers can be used to compare two or more arbitrary segments. Apart from having to match the feature specification, the segments in the input or output should also correspond to the requirements set by the identity markers. If the identity markers are the same, the segments should be the same; if the identity markers are different, the segments should differ. So, for instance, '`<+cons>i,<+cons>j`' is a pattern that denotes two consecutive consonants which are not the same.

Labels. Labels can be used to describe information which is associated with a particular segment, but which alters its nature. Lexical stress, for instance, is not an inherent feature of a vowel, but it sometimes is and sometimes is not associated with it. Labels can also be used to represent non-segmental information like word stress, but also word-class or sentence accent can be represented, for instance by labelling this information to the first segment of a word. Unlike features, labels are not necessarily binary. One may want to make a distinction between primary stress, secondary stress and no stress. In this case the label '`stress`' has values 1, 2 and 0 respectively.

In the user-typed input and the final system-provided output, the label information must be placed between the segments, since only a linear representation, a string, is available for input and output. Internally, the labels are aligned with the segments rather than placed between them. In this way, the segmental structure of a string is not distorted. That is, in a rule one does not have to take into account that a vowel might have accent, '`<+segm,-cons>`' selects the vowel whether or not there is stress associated with it. On the other hand one *can* access this information by specifying '`<* 1stress *>`'. If the vowel bears stress the primitive matches.

Before anything else happens, the label information present in the user-typed input string is extracted and aligned with the appropriate segments. Likewise, the last action is to insert the labels which are aligned with output segments, into the output string which is to be passed, for instance, to

consecutive modules of the text-to-speech system.

So, apart from defining the number of labels and their representation in linguistic rules, the linguist must also specify a code for each value the label can have for representation in the input and output. Thus, in the rules primary lexical stress is for instance denoted as '`<* 1stress *>`' and in the output with (for instance) an asterisk before the stress-bearing vowel: '`ex*ample`'.

Patterns

Primitives are in fact the most simple patterns, as they refer to exactly one segment in the input or output. More complicated patterns can be built by combining the primitives by means of *operators*, which in turn can also be combined by operators to form still more complicated patterns. TooIiP features five operators. In this respect the SPE formalism by Chomsky & Halle (1968) is somewhat modified and extended, tuned to the application of converting graphemes into phonemes.

1. The *concatenation* operator is denoted by the comma: '`,`', and is used to express sequential arrangement of patterns. It is placed between the primitives or patterns that must be found successively in the input or output, e.g., (2.3).

2. The *alternative* operator is denoted by pairs of curly brackets: '`{...}`', and is used to express an or-relationship between patterns. They are placed exactly below each other for each alternative, to adhere to the paper-and-pencil notation as closely as possible. Any number of alternatives can be specified. This operator is exemplified in (2.4), a simplified pronunciation rule for the 'c' in Dutch:

```
c  ->  S  /  _  {e}                    ! Cecilia        (2.4)
               {i}
```

A '`c`' should be pronounced as an /s/ if it is followed by either an '`e`' or an '`i`'.

3. The *optional* operator is denoted by parentheses: '`(...)`'. It operates on one argument, the pattern placed between the parentheses. It is used for repetitive patterns, and the parentheses may be followed by the minimum and maximum number of times the structure should be present. Examples are:

`(<+CONS>)2-5`	=	A minimum of two and a maximum of five phoneme consonants.
`(<-cons,+segm>)0`	=	Zero or more grapheme vowels.
`(A)`	=	An optional (zero or one) phoneme '`A`'.

4. The *complementation* operator is denoted by an apostropghe: '', and is used to express the absence of a pattern. It operates on the first primitive or structure following the quote:

```
c  ->  K  /  _  'h                    ! colbert        (2.5)
```

A 'c' is pronounced as a /k/ if it is not followed by an 'h'.

This operator is not present in the SPE formalism, but is included for elegant rule description. For instance, exceptions to rules can well be treated with this mechanism. It turns out there are some logical problems connected with its interpretation in certain structures, but these are treated extensively elsewhere (Van Leeuwen, 1987; this thesis, chapter 3).

5. The *simultaneity* operator is denoted by angled brackets: '[...]', and is used to express an and-relationship between patterns. Like the alternative operator, it can have any number of arguments, and each argument, enclosed in brackets, is placed beneath the other. This operator, too, is not present in the SPE formalism, and is included for elegant rule description. The operator is typically used for two purposes. One is to intersect sets, for instance:

```
[ <+cons> ]                                          (2.6)
[   'c    ]
```

The set of all grapheme consonants is intersected with the set of all graphemes except 'c', so the structure denotes "any grapheme consonant except 'c'".

The other use of the operator is to express alignment between graphemes and phonemes as in rule (2.7):

```
[ OO ]  ->  <* 1stress *> /  _  <-segm>  ! niveau      (2.7)
[e,a,u]
```

The label primary stress is assigned to the phoneme 'OO' if it is derived from the orthography 'eau' and located at the end of the word.

In this way one can distinguish elegantly between the same phonemic representations which have different underlying orthographic structures.

With these operators, a user can define any pattern of segments to his or her liking. With the first two operators, the concatenation and the alternative operator, in principle any finite pattern for the input or output string can be constructed. Because the number of possible segments is finite, any set of segments can be composed with the alternative operator by means of enumeration. Strings can be composed with the concatenation operator, so sets

of finite strings can be composed with the combination of the two. For repetitive patterns the optional operator is needed. The simultaneity operator is necessary to express alignment of graphemes and phonemes. The complementation operator does not enhance the power of expression of the formalism, but serves well for elegant and transparent pattern description. The alternative pattern for (2.6), where the complementation is used to exclude the 'c', would be an extensive alternative structure, which would need closer study to reveal its meaning, whereas a glance at (2.6) is sufficient.

Actions

When all the patterns of a rule match, the specified action is performed. The structural change is added to the output and aligned with the focus pattern. For the structural change only (possibly concatenated) primitives can be substituted, no other operators are allowed. Therefore, three corresponding types of action are distinguished: *segment assignment*, *feature modification* and *label assignment*.

Segment assignment. The most commonly used mechanism is to assign segments in the output to segments in the input. The alignment of input and output is represented by vertical lines. Consider for instance rule (2.8):

```
c,h  ->  SJ /  _  a,u                    ! chauffeur        (2.8)
```

and suppose that the internal state is as follows:

```
input:  | c | h | a | u | f | f | e | u | r |
        | ↑ |
output: |   |                                              (2.9)
```

The arrow is a *reference marker* that points at the input segment being dealt with. As can be seen, the patterns of (2.8) match, so the 'SJ' will be added to the output and aligned with the grapheme sequence 'ch'. This is reflected in (2.10) by the altered vertical alignment. The segments 'c' and 'h' should no longer be considered separately, but as a sequence which as a whole is aligned with the 'SJ':

```
input:  | c h | a | u | f | f | e | u | r |
        |     | ↑ |
output: | SJ  |   |                                        (2.10)
```

In principle all segment manipulations can be formulated with such segment assignment rules, although elegant and concise rules will not always

result. For this reason, another mechanism has been introduced, feature modification, mainly to be able to capture phenomena in one rule that would otherwise require many similar rules.

Feature modification. Feature modification rules deal with phonological generalizations. A well known example is the phenomenon that in a number of Germanic languages word-final obstruents become voiceless. This is expressed elegantly in (2.11), where the obstruents are selected by the feature '<-SON>':

```
<-SON>  ->  <-VOICE>  /  _  <-SEGM>                    (2.11)
```

Suppose that at some time the following state is reached:

input:	...	H	U	D	␣	...
				↑		
output:	...	H	U			

(2.12)

and that 'D' is defined as '<-SON>' and ␣ (space) as '<-SEGM>'. The rule applies, so the features (in this case there only is one) in the structural change replace the corresponding original, which results in a new feature bundle. The corresponding segment will be searched for in the segment definition table, and added to the output. If no such segment is found, a special 'error-segment' is added, and an error message is sent to the user. After the application of the rule, the internal state will be as follows:

input:	...	H	U	D	␣	...
					↑	
output:	...	H	U	T		

(2.13)

Label assignment. Label assignment rules assign labels to segments. The labels are aligned with segments, and thus have a separate (parallel) representation level. The segmental structure of the input and output thus remains unaltered, so that later rules, only operating on the segmental level, will not be bothered by the labels. Consider rule (2.14):

```
[    OO    ]  ->  <* 1stress *>    ! Bordeaux, cadeau     (2.14)
[e,a,u,(x)]
```

If this rule applies, the label '1stress' is set on the label representation level, aligned with the phoneme 'OO' and the grapheme string 'eau' or 'eaux'. The internal state then will be as follows:

input:	c	a	d	e a u	
output:	K	AA	D	00	(2.15)
labels:	-	-	-	1stress	

The system output which the user receives if 'cadeau' is typed in will be: 'K AA D *00' (the spaces separate the phonemes). The label information, if present, is inserted before the phoneme it is aligned with. In this case an asterisk is the output representation of primary stress.

2.3.2 Modules

Thus, with linguistic rules one can transcribe an input segment into an output segment. To transcribe an input string into an output string one needs to group a set of rules into a module. A module is the smallest unit that takes a string as input and produces a string as output. This section deals with what a module looks like. First the general assignment scheme is discussed, i.e., which procedure is used to determine the output string from the input string given the specified rules. Then the grouping of rules into blocks and how these blocks are consulted is discussed. Finally, an example is given of how this works in practice.

Assignment scheme

The input of a module is the module's input buffer. The input buffer is filled with an input string which is surrounded by a number of spaces. The spaces serve to provide a neutral left- and right context for the leftmost and rightmost segments respectively. The output of the module is written into the module's output buffer.

The input buffer is scanned once. For each module the linguist can choose whether this should be from left to right or from right to left. For instance, stripping suffixes can be done elegantly if the input string is scanned from right to left, while prefixes are best handled from left to right. Scanning in the scanning direction, the input segments are considered in order. Simplifying somewhat, for the current input segment the rules are consulted from top to bottom, until a rule matches. The structural change of the rule is added to the output and aligned with the corresponding input, and the remainder of the rules is skipped. Then, dependent on the length of the focus the next input segment is 'selected', and the procedure is repeated. This procedure is called *segment-by-segment assignment*, as the segments are processed one by one.

If none of the rules match, it depends on the nature of the module what happens. Either the input segment is simply copied to the output and aligned, or an error condition occurs. This is addressed in a subsequent section, but in either case the next segment is selected, and thus the procedure of transcribing a segment always operates on the first still unprocessed input segment.

Rule types

Due to the segment-by-segment assignment strategy, a user can order the rules according to the segment on which they operate, and thus improve efficiency. For instance, all rules concerning the character 'c', such as (2.3–2.5), can be grouped into a rule block which is only consulted when a 'c' is encountered in the input. The rules within a block are consulted in the top to bottom manner, with the first rule that matches applying. This is essentially an 'if then else if...' construction. The rule blocks, on the other hand, are mutually exclusive. The rule blocks operate as a 'case statement', the input segment being the case selector.

Not all rules, however, can be grouped in this way, because some rules have no triggering segment, while others are triggered by more than one segment. Rules of the first type are called *insertion rules*, those of the second type *common rules*. Rules of the type which are triggered by exactly one segment are called *segment rules*.

An example of an insertion rule is given in (2.16):

```
0 -> + / ... _ l,y                                    (2.16)
```

Here, a '0' (zero) denotes an empty focus. An affix marker '+' must be inserted before the affix 'ly' (under circumstances in the left context which are left unspecified here).

Since the focus of an insertion rule is empty, the rule cannot be triggered by a specific segment, and must therefore be tested for each position in the string.

An example of a segment rule is given in rule (2.8). In left-to-right scanning mode the rule is triggered by a 'c'; only when the segment 'c' is encountered in the input does this rule have to be consulted. Rule (2.11) is an example of a common rule, which is triggered by any phoneme defined as '<-SON>'. This applies to more than one segment, so the common rules, too, must be consulted for each segment in the input string.

Due to this distinction in rule types the syntax of a module is as follows. First, all rules must be grouped in the appropriate block: the insertion rules in the insertion block, the segment rules in the appropriate segment blocks and the common rules in the common block.

For each input segment first the insertion block is consulted from top to bottom. If an insertion rule matches, the structural change is added to the output and the remainder of the insertion rules is skipped. If no insertion matches nothing changes in the output. The input segment has not been processed in either case. For the same input segment the segment block is now consulted. The segment block is the case statement of rule blocks for the various characters. The input segment selects the appropriate rule block, and consults it from top to bottom. If a rule matches, the remainder of the rules—including the common block—are skipped and the next input segment is dealt with. Only if no segment rule matches will the common block be consulted. This is consulted in the same top to bottom manner. Only if no common rule matches will the 'copy or error' action take place.

The procedure of first consulting the insertion block, then the segment block and finally the common block is induced by the segment-by-segment strategy. It does not, however, constrain the linguistic possibilities. If one needs to alternate insertion and segment rules, or segment and common rules, one can introduce separate modules for each of these blocks. By concatenating them in the desired order one can obtain the desired function.

An example

As an example the conversion of the first part of the word '`chauffeur`' is treated. The relevant rules, which are a small sample of a grapheme-to-phoneme module, are included in Table 2.I.

Suppose the input is scanned from left to right, and initially the internal state is as in (2.17), the reference marker being positioned at the first character, '`c`':

input:	c	h	a	u	f	f	e	u	r
	↑								
output:									

(2.17)

Since there are no insertion rules, the character rules are consulted directly. As the reference marker is positioned at a '`c`', the rules under '`grapheme c`' are selected. The first rule fails to match as neither an '`e`' nor an '`i`' follows the '`c`'. The second rule fails as the right context of that rule requires any character except an '`h`' and that is exactly what is found. The third rule matches; first the focus '`ch`' is found, and directly to the right of the focus '`au`' is found. The rule applies; the structural change '`SJ`' is added to the output and aligned with the focus '`ch`'. The reference marker is shifted two characters to the right, as the focus consists of two characters:

```
input:  | c h | a | u | f | f | e | u | r |
                ↑
output: | SJ  |                                   (2.18)
```

The new reference marker points at 'a', so now the rules for the grapheme 'a' are selected. The first rule fails on the focus, as 'au' is found instead of 'aa'. The second rule matches, however. The focus matches, and to the left of the focus the phoneme 'SJ' is found. So 'OO' is added to the output and aligned with 'au'. The reference point is shifted along the length of the focus:

```
input:  | c h | a u | f | f | e | u | r |
                    ↑
output: | SJ  | OO  |                             (2.19)
```

2.3.3 Conversion scheme

With a module one can manipulate strings. Often a particular conversion can be divided into several sequential steps. For this purpose the user can concatenate modules; every next module operates on the output of the previous one. Together the modules define a conversion scheme.

Table 2.I: Part of a module, which is used for the conversion of the first part of the word 'chauffeur'.

```
grapheme a                          ! Rules for Dutch
==========
a,a  ->  AA  /  _   'u              ! aap, not blaauw
a,u  ->  OO  /  SJ  _               ! chauffeur
a,u  ->  AU                         ! normal 'au' rule
:
grapheme c
==========
c    ->  S   /  _   {e}             ! Cecilia, not cacao
                    {i}
c    ->  K   /  _   'h              ! colbert, not chauffeur
c,h  ->  SJ  /  _   a,u             ! chauffeur
:
```

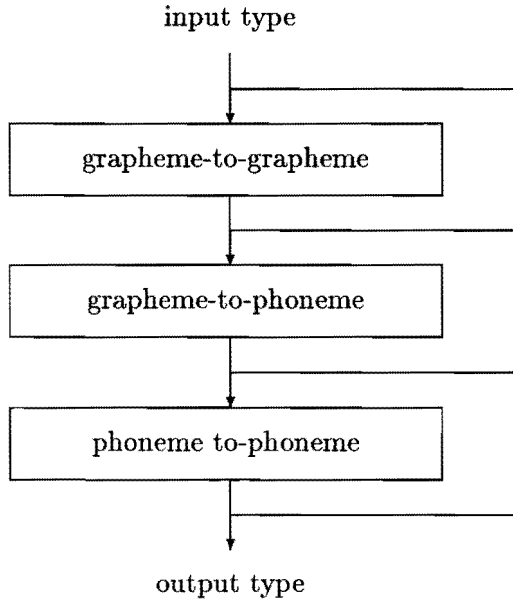


Figure 2.3: The ways in which modules may be concatenated.

As holds for the total conversion, the input and output of separate modules can either be of the same type or not. A natural constraint is that the output and input types of consecutive modules should be compatible. As many modules as necessary may be concatenated.

The notion of two separate levels, an input and an output level, has been introduced in order to provide access to information concerning the relation between spelling and pronunciation. Therefore, input-types (graphemes) and output-types (phonemes) must not be mixed, and, as a consequence, the transition from the input to the output type may be made only once[6]. This is depicted in Fig. 2.3.

Once the (optional) transition to phonemes has been made, one can refer to both graphemes and phonemes in the rules, along with their alignment. Conversely, referring to phonemes when only grapheme modules have been used is not possible, of course. This opens the possibility to interpret output symbols (symbols which generally denote phonemes) within these modules alternatively.

For instance, an elegant way to distinguish graphemes from phonemes—given one wants to do so—is 'case coding': a 'k' is a grapheme, 'K' a phoneme.

[6]Although the input-types and the output-types are not restricted to graphemes and phonemes only, graphemes and phonemes will denote them respectively in the remainder of this section.

For a grapheme-to-phoneme module this works well, but it implies that one cannot deal with capitals.

Therefore, in the first phase of the conversion, when only grapheme modules are used, the phoneme segments (for instance those which are denoted by a single capital letter) can be used as if they were grapheme segments. In this way one can write a de-capitalising module, which operates after a module that deals with acronyms (abbreviations like 'UK' and 'BBC'). An obvious demand is that when the grapheme-to-phoneme module is being consulted, uppercase characters should be present in the input.

Now the 'copy or error' action can be specified which TooɪjP performs if for a certain segment none of the rules in a certain module match. In modules that have the same type of segments as input as they produce as output, the segments for which no rules apply are simply copied to the output. In the grapheme-to-phoneme module no copy action can be taken, of course. Here, a harmless error segment is added to the output to mark the position where the error occurred, and the error condition is reported to the user.

This also implies that TooɪjP is robust: all input can be handled, including unpronounceable input. The system simply executes the specified rules. This always results in an output string, possibly containing error segments.

2.4 System output

Given the deterministic way the rule base is consulted, the conversion scheme contains all the information for the conversion explicitly. On the other hand, it contains only this type of information, it does not include a mechanism to actually perform the conversion. This bipartition, depicted in Fig. 2.4, gives the system a maximum of flexibility: any conversion scheme for the manipulation of strings specified in the linguistic section can be executed. In this way, an efficient division of labour and field of speciality has been achieved. A linguist formulates, for instance, the rules for grapheme-to-phoneme conversion, while an engineer develops the machine which performs the specified conversion. Their interface is the interpretation and consultation scheme of the rules.

The conversion scheme is presented to TooɪjP in text files. Each module is a separate text file. A specific text file indicates the order in which the modules are to be consulted. The definition tables of the segments, features and labels are also included in separate files.

While the string of output segments is, naturally, TooɪjP's most important output, the system is also capable of providing information on the conversion

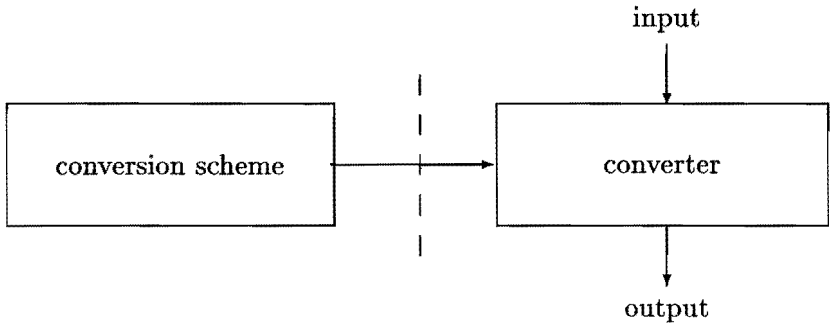


Figure 2.4: The (linguistic) conversion knowledge is separated from the mechanism which performs the conversion.

process and gathering information for statistical purposes. For development purposes debugging tools have been provided for, to facilitate the tracing of errors in the rules. When conversion performance is satisfactory, the linguist may want to know how frequently the rules apply, to try and improve the conversion speed. For this purpose a rule coverage analysis tool has been included. Finally, when the set of rules has been developed TooIJP can be used to collect statistical information on grapheme-to-phoneme correspondences, as these are being preserved in the system. These three types of output will now be described in greater detail.

2.4.1 Development Support

In the conversion scheme three levels can be distinguished. The conversion scheme consists of modules, a module consists of linguistic rules, and the rules consist of patterns and actions. In most cases, the number of modules will not be extremely large. For instance, we have built a complete grapheme-to-phoneme conversion system for Dutch with only some 8 to 12 modules (see Berendsen *et al.*, 1986). The number of rules to be consulted in succession, however, can become large: we have a module with over 150 rules. Also, the complexity of the rules can be high, when operators are nested to great depths. Therefore, powerful debugging tools can be useful. TooIJP provides three corresponding display modes to help debugging. The first one displays the output of each module, the second one displays the performance of the rules within a specific module, and the third one displays the matching process within specific rules.

2.4.2 Rule Coverage Analysis

A second development tool which has been included in TooLjP is a so-called rule coverage analysis. This feature can keep track of the frequency with which individual rules are consulted, and determine whether or not they have been applied. It may turn out that quite a number of rules hardly ever apply, for instance because they have been designed for words of very low frequency. In that case the linguist may want to rearrange the rules in such a way that the rules for infrequent regularities are tested last. This can help to improve the performance speed of a module, and consequently of the conversion scheme.

2.4.3 Derivation Analysis

As mentioned in earlier sections, TooLjP can be used as an analysis tool to collect statistical information on grapheme-to-phoneme correspondences. This is possible because during the conversion of a word or a sentence the derivational history is retained. Not only are the results of each module retained, but also the individual input-to-output relations.

There are two types of information on derivational history. The first type is a detailed report on the derivational history of specific input. The second type is an overview of the grapheme-to-phoneme relations of accumulated input.

The derivational history report looks much like the internal state figures as in (2.18) and (2.19), only here all the modules are included. Given a certain scheme of concatenated modules, a typical example is shown in (2.20), where a full derivation of the word '`chauffeur`' is given.

input:	c h	a u	f	f		e	u	r		
morphology:	c h	a u	f	f	#	e	u	r		
graphon:	SJ	OO	F	F	MB	EU		R		(2.20)
stress:	SJ	^OO	F	F	MB	*EU		R		
reduction:	SJ	^OO	F			*EU		R		

Here, a '`#`' is the morpheme boundary marker on the grapheme level, '`MB`' is the morpheme boundary marker on the phoneme level, an asterisk denotes primary stress and a hat denotes secondary stress. Vertical lines indicate synchronization of segments

between modules. So, for instance, in the morpheme module a morpheme boundary was inserted between the 'f' and the 'e', and in the graphon module the graphemes 'c' and 'h' are associated with the phoneme 'SJ'.

From this detailed derivational history report the individual grapheme-to-phoneme relations are computed. Each smallest group of input segments that is synchronized with the output is stored in the derivation database with its synchronized output. Synchronization means that the vertical synchronization marks are extended from input to output on both sides of the segments. Thus in (2.20) the 'c' at the input level is not synchronized with the output as the right synchronization mark stops at the graphon level. In the example 'ch' is synchronized with 'SJ', 'au' with 'OO', 'ff' with 'F', 'eu' with 'EU', and 'r' with 'R'. These five grapheme-to-phoneme relations are stored in the derivation database.

For each conversion, grapheme-to-phoneme relations can be computed. If a new relation is found, a new entry is created in the database. If a relation is found that has already occurred, the number of occurrences is incremented. When all input has been processed and the grapheme-to-phoneme relations are included in the database, the results are printed alphabetically. For each group of input segments that has been stored, the corresponding output(s) are listed with their frequency. An example is given in Table 2.II.

2.5 Extensions

In the previous sections only the essential characteristics of TooɟiP have been described. Some additional features were omitted, which are described in this section.

2.5.1 Meta-symbols

Some symbols have a special meaning in the rules. For instance, a single '0' in the focus or structural change denotes an insertion or deletion rule, and '/' indicates the beginning of the context specification. Such symbols are called meta-symbols, as they have the syntactic function to denote the structure of a rule.

Because of this function, they cannot be used freely to refer to segments in the input or output. Therefore, a mechanism has been provided to define a symbolic name for a character, so that the characters that correspond to the meta-symbols can be referred to. In the definition tables of input and output segments one can include such definitions, for instance:

```
zero = 0                                                    (2.21)
```

Now, when the character sequence '`zero`' is used in the rules, it will refer to the segment '`0`' in the input.

2.5.2 Macro Patterns

In the linguistic rules, some patterns are used very often. For instance, '`<-CONS,+SEGM>`' is such a pattern, denoting all vowels. It may be desirable to introduce a shorthand notation for such patterns. These are called *macros*. They not only reduce the typing effort, but also make the rules easier to read.

At the beginning of each module one can define the macros valid for that module, as exemplified in (2.22):

```
VOW = <-CONS,+SEGM>          ! vowels                       (2.22)
```

Each time this macro is encountered, the pattern associated with it replaces the macro. Previously defined macros can be defined in the patterns of new ones.

Table 2.II: Derivation analysis of the grapheme '`c`' in a random word list. Listed to the left of the arrow are the input graphemes, between parentheses the number of occurrences. Listed to the right of the arrow are the alternative pronunciations with their relative frequency.

```
⋮
grapheme c
----------
c  (671) ->  K   78.5 %    S   21.5 %
cc  (10) ->  K  100.0 %
ch (153) ->  K    1.3 %    SJ  30.7 %    TSJ  1.3 %
             X   66.7 %
ck  (11) ->  K  100.0 %
⋮
```

2.5.3 Metathesis

Metathesis is the exchange of two or more segments. This mechanism is typically needed for elegant text normalization of Dutch or German numbers, amounts of money and indication of time. In these languages, the digits of the numbers between 13 and 99 are pronounced in inverse order. Take, for instance, the number '36', which is pronounced as "zes en dertig" (six and thirty). With the facilities described so far one would have to specify some 90 rules of the type:

```
3,6  ->  6,3  /  ... _  ...                    (2.23)
```

With metathesis one can capture this phenomenon in one feature rule:

```
<+num>i,<+num>j  ->  <+num>j,<+num>i  /  ... _ ...    (2.24)
```

Just like normal rules using identity markers, matching the focus pattern of (2.24) to the input string results in storing the segments found. If the rule matches, the segments are retrieved and added to the output in the specified order.

2.5.4 Exception lexicon

As stated in the introduction, rule-based grapheme-to-phoneme conversion will always need an exception lexicon to cover irregularities. For this reason TooIjP has been extended with the possibility to include an exception lexicon. The presented orthographic input is decomposed into words, which subsequently are looked up in the lexicon. If a word is present, the corresponding phonetic transcription is assigned. If it is not present, it is transcribed by rule. The output of lexicon-lookup and rule-transcription is then merged. Before and after this lexicon some segment processing can be done. For instance, before the lexicon lookup some text normalization may be needed, and afterwards some inter-word processes such as assimilation may be desirable. These processes are formulated in the same way as the rule-based conversion, by means of linguistic rules in modules. This is indicated schematically in Fig. 2.5.

The lexicon was constructed by Lammens (1987) in such a way that it can easily be implemented in other systems. The user can insert and delete new entries, which are coded in the same format as TooIjP's input and output.

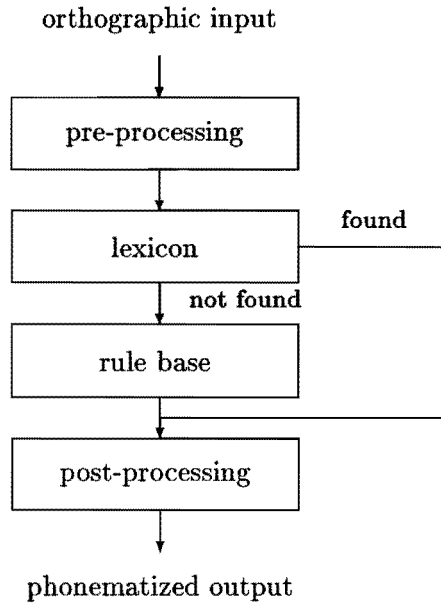


Figure 2.5: Structure of the development system when an exception lexicon is included. The pre-processing, the rule-base and the post-processing are all conversion schemes.

2.6 Relation to other systems

2.6.1 Lay-out

In most systems, the linguistic data to which a rule has access are represented on one level only, viz. the output of the previous module or rule. In TooIjP, more levels of linguistic information can be consulted. The input-type level (graphemes) and the label level are always accessible, the output-type level (phonemes) is available after it has been introduced. As each segment has a unique representation, there is no doubt as to which level is referred to in the rules. Information on the label level is accessed by enclosing the desired label information in special brackets, as described in earlier sections.

The Delta system of Hertz, Kadin & Karplus (1985) is even more flexible: the linguist can define any number of levels. A disadvantage is that in order to refer to these levels in the linguistic rules, use must be made of special selection markers, as the segments have the same representation on different levels. These selection markers somewhat complicate a quick understanding of the rules.

A second aspect of lay-out is the way in which or- and and-relationships are represented. Normally, this is done one-dimensionally, so that the structure can be parsed in one scan. A typical representation is given in the insertion rule (2.25). The purpose of the rule is to insert a prefix boundary marker behind the Dutch prefixes 'be', 'ge', 'her' and 'ver'. The right context is omitted here. The left context is an alternative structure with two options, each of which contains another alternative structure. The first alternative accounts for 'be' and 'ge', the second one for 'her' and 'ver'. The slash within braces denotes the separation of the alternatives:

```
0  ->  +  /  {{b/g},e/{h/v},e,r}  _  ...          (2.25)
```

TooIjP features a two-dimensional representation of or- and and-relationships, which increases the readability of the rules. It requires more space and a more complicated parsing strategy, but offers the linguist greater insight into the nature of the patterns than a one-dimensional representation. Compare, for instance, (2.25) and (2.26). In (2.26), the same rule as (2.25) is represented in the two-dimensional representation.

```
0  ->  +  /  { {b},e }  _  ...                    (2.26)
               {g}
             {{h},e,r}
              {v}
```

The idea behind this two-dimensional representation is that it reflects the structure of patterns better than a one-dimensional representation. Horizontal positioning of patterns reflects a sequential relationship; concatenated structures should be found in succession. Vertical positioning reflects that the patterns refer to the same position of the input or output string. The linguist is able to use more complicated patterns owing to this way of representing them, and it has been observed that in practice this possibility is being used extensively.

2.6.2 Ordering principle

Inside each module and within each type of rule, the consultation of the rules is order-sensitive. In most grapheme-to-phoneme conversion systems this is the case. The main reason for this is that linguists are used to working with sets of ordered rules. This implies that one cannot evaluate the performance of an isolated rule, one can only judge the whole set of rules. In principle, this is a disadvantage. However, compared to systems in which the rules are

unordered and therefore disjunct (e.g., Van der Steen, 1987) the individual rules in an order-sensitive system are less complicated, as each next rule deals with a decreasing number of cases. Given the number and complexity of the rules in a realistic application, this turns into an advantage over systems where rules are unordered.

2.6.3 Assignment strategy

In several other systems (Carlson & Granström, 1976; Hertz, 1982; Holtse & Olsen, 1985), the assignment strategy differs from the module-internal strategy defined here. Instead of applying a segment-by-segment strategy—consulting all appropriate rules from top to bottom for a certain input segment—a rule-by-rule strategy is applied: one rule operates on the complete input string, the modifications brought about being input to the next rule. A rule then has string-manipulation characteristics: it modifies all appropriate segments of an input string.

TooLjP features both strategies: within a module the segment-by-segment strategy is applied, while the string-manipulation strategy is applied by succession of modules. Both strategies can be useful. The system described by Kerkhoff *et al.* (1984), for instance, also features both strategies, but in that system a rule-by-rule strategy is chosen by default.

The difference between a segment-by-segment and a rule-by-rule strategy obviously has consequences for the way in which linguistic rules should be formulated. Although TooLjP does not feature an explicit rule-by-rule strategy, this can be simulated by including only one linguistic rule in a module, and defining just as many modules as there are rules. Therefore, TooLjP has at least the same possibilities as the others. However, by including more than one rule in a module and organizing them according to the segments on which they operate, efficiency is improved, since the rules are tested only on the appropriate segments. As some 80–90 % of the rules in an actual application of grapheme-to-phoneme conversion are segment rules, the increase in efficiency is considerable.

2.7 Applications

No matter how well-designed a system is, the final test that must be applied is: "Is it useful in practice?". So far, the main test for TooLjP has been the development of a grapheme-to-phoneme conversion system for Dutch (Berendsen *et al.*, 1986). The conversion scheme comprises the full trajectory from unrestricted orthographic text to phoneme transcription, complete with word

stress markers. The results appear to be promising: a 96.5 % correct transcription score was obtained for 4000 word types in an arbitrary newspaper text (Berendsen, Lammens & Van Leeuwen, 1989).

In developing the rulebase, TooIiP's facilities have proved to be satisfactory. The tools to construct a rulebase have appeared to be adequate, elegant and flexible. The debugging tools in particular have served well in the development of the rules.

The grapheme-to-phoneme converter now has two direct applications. On the one hand, it serves as a research tool to collect statistical information on the individual grapheme-to-phoneme relations for Dutch, and on the other, it serves satisfactorily as part of a text-to-speech system for Dutch.

Apart from these scientific applications, TooIiP is also being used for educational purposes. It appears to be well suited for a quick verification of phonological descriptions and it serves as a practical teaching-aid to illustrate and practice the use of SPE-like linguistic rules.

2.8 Conclusion

In this chapter the TooIiP system has been described for a user's point of view. It serves as a development tool for linguistic rules. The linguistic rules can be ordered in a certain scheme so as to define the conversion of an arbitrary input string to a desired output string. The rules contain the information on how the conversion is to be executed, and the system performs the conversion, driven by the rules.

A special characteristic of TooIiP is that input and output can be of a different type, and that the derivation of input to output is retained. Due to this, one can refer in the rules to this alignment of input and output. When the system is used as a grapheme-to-phoneme conversion system, this alignment reflects the correspondence of spelling and pronunciation. As a consequence, once a grapheme-to-phoneme conversion scheme for a certain language has been developed satisfactorily, spelling-to-sound correspondences in that language can be studied by using TooIiP as an analysis tool to collect and analyse such data.

Appendix 2.A
Functional specification of TooIJiP's main body

In this Appendix a functional specification is given of TooIJiP's main body, as it is described in section 2.3. The specification will be given on a rather high level, thus abstracting from irrelevant detail. The primary goal is to define formally how linguistic rules are evaluated, how a rule block is evaluated, and how a module and the conversion scheme are executed. Therefore, some lower-level functions are not specified formally but described in natural language.

Data structure

$$\left.\begin{array}{ll} \mathit{input} & : \\ \mathit{graph} & : \\ \mathit{phon} & : \\ \mathit{output} & : \end{array}\right\} \textbf{array } [1..\mathit{len}] \textbf{ of segment}$$

The main data structure consists of 4 arrays. *input* and *output* serve to store the original input and the system-provided output and are also necessary for overall synchronization purposes. The 'real work' is being done in *graph* and *phon*.

Functionality

function $P\_match(P, match\_dir, start\_pos) =$
$Gen\_string(start\_pos, match\_dir) \cap \langle P \rangle \neq \emptyset$

P_match: The basic function is to match a pattern to the data buffers. *P_match* is defined in terms of *Gen_string* and $\langle P \rangle$. $\langle P \rangle$ is the set of string denoted by P, on which chapter 3 elaborates. *Gen_string* is a function that generates all strings of segments which can be formed by starting at *start_pos* and concatenating all segments that are encountered between *start_pos* and the end of the buffer in the matching direction *match_dir*, where at the synchronization points between *graph* and *phon* one may switch from one buffer to the other.

function $R\_match(F, L, R, scan\_dir, int\_pos) =$
  if $(scan\_dir = \rightarrow)$
  then $P\_match(F, \rightarrow, int\_pos) \wedge$
    $\exists l \in Length(F) : P\_match(R, \rightarrow, int\_pos + l) \wedge$
    $P\_match(L, \leftarrow, int\_pos - 1)$
  else $P\_match(F, \leftarrow, int\_pos) \wedge$
    $P\_match(R, \rightarrow, int\_pos + 1) \wedge$
    $\exists l \in Length(F) : P\_match(L, \leftarrow, int\_pos - l)$
  fi

R_match: F, L and R denote focus, left and right context respectively. *scan_dir* is the scanning direction. *int_pos* is the internal position (see section 4.4), this denotes the position in *work_buf* up to where the transcription process has proceeded. *Length* is a function that generates set of lengths for which the (focus) pattern matches. For instance, $Length(\{^{\mathbf{a}}_{\mathbf{a,t}}\}) = \{1,2\}$ if the input contains 'at' at the appropriate position, but it is $\emptyset$ if 'b' is found.

function *Apply_rule*($Rule_i$) =
  if *R_match*(F_i, L_i, R_i, *scan_dir*, *int_pos*)
  then
    if (*gra*→*phon*) **then** "Add C_i to *phon*"
      else "Replace F_i by C_i in *work_buf*" **fi**;
    if (*gra*→*phon*) **or** (*phon*→*phon*)
      then "Synchronize C_i(*phon*) with F_i(*graph*)" **fi**;
    "Synchronize *work_buf* with *input*";
    "Advance *int_pos* by *Length*(C_i) in *scan_dir*";
    return true
  else
    return false
  fi

Apply_rule: F_i, L_i and R_i are the focus, left- and right context of $Rule_i$. *gra*→*phon* denotes that the rule is part of a grapheme-to-grapheme module. The other two possibilities are *gra*→*gra* and *phon*→*phon*. The bulk of the output is generated by this function. It is given as a side effect rather than the result of the function. Synchronization, too, is a side effect.

function *Apply_Block*(*Rule_block*) =
  $i := 1$;
  success := **false**;
  while not *success* **and** *Present*($Rule_i$) **do**
    success := *Apply_rule*($Rule_i$);
    $i := i + 1$
  od;
  return *success*

Apply_block: Starting with the first rule, the rules are consulted until a rule matches. If a rule has been applied, *success* = **true**, otherwise it is **false**.

procedure *Apply_Module*(Mod_i) =
  if (*scan_dir* = →) **then** *int_pos* := 1
    else *int_pos* := *len* **fi**;
  while (*int_pos* ∈ *Input_range*)

```
do
    if Present(Insertion_block_i)
        then result := Apply_Block(Insertion_block_i) fi;
    if Present(Segm_block_i)
        then result := Apply_Block(Char(int_pos)_i) fi;
    if not result and Present(Common_block_i)
        then result := Apply_Block(Common_block_i) fi;
    if not result then
        if (gra→phon) then "Add ES {error segment} to phon" fi;
        "Advance int_pos by 1 in scan_dir"
    fi
od;
return true
```

Apply_Module: *Input_range* determines whether *int_pos* is still in the range of the input string. Depending on the scanning direction *int_pos* is initialized to the leftmost or rightmost segment. As can be seen, the segmental block is consulted independent of whether the insertion block has applied. The common block, on the other hand, is only applied when the segmental block has not applied.

```
procedure Apply_Conv_Scheme(Scheme)
    if (phon→phon) then phon := input
                   else  graph := input fi;
    i := 1;
    while Present(Mod_i)
    do
        if (phon→phon) then work_buf := phon
                       else  work_buf := graph fi;
        Apply_Module(Mod_i);
        i := i + 1;
    od;
    if (gra→gra) then output := graph
                 else  output := phon fi;
```

Apply_Conv_Scheme: *work_buf* is initially filled and defined for each module. The modules are applied in order. (*gra→gra*) and (*phon→phon*) are tests that concern the type of the overall conversion scheme.

Chapter 3

Extending regular expressions with complementation and simultaneity

Abstract

Regular expressions are a well-known mathematical tool in computer science. The patterns which are used in TooljP are, in fact, an extended form of regular expressions. For user convenience two operators are added to the standard regular expressions: complementation (the 'not') and simultaneity (the 'and').

The introduction is not without problems, however. If the complementation operator is introduced in a compositional manner, the formal interpretation of a certain class of expressions differs from what one would expect those expressions to mean. To be precise: certain strings one would expect to be excluded, are not. This is considered to be an undesirable characteristic, as generally users simply start using a system rather than first studying its exact nature.

Therefore, an alternative definition for complementation is proposed, which for the mentioned class of expressions behaves in accordance with expectation. The essential difference with the compositional formalism is that now the 'explicit nofits' are always excluded. As a consequence, however, strict compositionality is lost, which for instance shows in the fact that double complementation may not always be annihilated.

Next, the simultaneous operator is included, symmetrical to the alternative operator, which introduces a small additional complication. From a theoretical point of view the proposed formalism is not completely satisfactory. It might be satisfactory, however, from a practical point of view. Those patterns for which it behaves unsatisfactorily are highly unlikely to be used in practice, and the proposed formalism can be seen as a practical compromise between the practical needs and theoretical elegance. On these practical grounds it has therefore been decided to implement the semi-compositional formalism in TooljP.

3.1 Introduction

REGULAR expressions are a widely used mathematical tool in computer science. In general, they can be used to describe a certain class of (formal) languages, called regular sets, the characteristics of which can be found in any text book on this subject (see for instance Hopcroft & Ullman, 1979). Regular expressions are used in a wide variety of applications, one of which, for instance, is rule-based grapheme-to-phoneme conversion (see for instance Carlson & Granström, 1976; Hertz, 1982; Holtse & Olsen, 1985; Kerkhoff, Wester & Boves, 1984; Van Leeuwen, 1989). In such an application the conversion of orthographic text (the graphemes) into a sound representation (phonemes) is controlled by a set of linguistic transcription rules, which are often a particular version of Chomskian rewrite rules (Chomsky & Halle, 1968). Each rule is a recipe on how to rewrite certain input characters, given their presence in a certain context. The specification of the characters to be transcribed, and the context in which the transcription should take place, can well be done by means of regular expressions, as the type of strings one needs to denote for this purpose generally fall within the class of regular sets. Often some small extensions are made to this basic formalism, tuned to the specific requirements of the application.

Apart from the so-called terminals, which directly refer to the characters of some alphabet, regular expressions feature three operators for the specification of more complex regular expressions. Alternation is used to obtain the union of two regular sets, concatenation is used to express juxtaposition of two regular sets, and repetition is used for infinite concatenation of regular sets. In this chapter the problems are discussed that were encountered when two operators, the complementation operator (the 'not') and the simultaneous operator (then 'and'), were added to the standard regular expressions.

The main motivation for this extension was to increase the user's ease of expression. With the thus extended regular expressions target and context patterns are specified in the linguistic rewrite rules of the TooLiP system. With it one can develop a set of linguistic rules, which, for instance, define the grapheme-to-phoneme conversion for a specific language.

Several other systems which feature similar rules for similar purposes (e.g., Hertz, Kadin & Karplus, 1985; Kerkhoff *et al.*, 1984; Van der Steen, 1987) also feature a complementation operator in their own versions of extended regular expressions, and each system features its own interpretation, without being too clear, however, on the exact interpretation and the underlying motivation for that specific interpretation. This chapter aims to identify and propose a solution to the problems attached to this matter.

Another, totally different, application where the problem may also be encountered concerns the search function of a text editor. Simple search functions only allow a sequence of characters as search pattern. More sophisticated functions also feature 'or' structures, thus for instance one can search for a 'c' followed by an 'e' or an 'i'. When a 'not' structure is also introduced (for instance, "search for a 'c' not followed by an 'e' or an 'i'") the search function has essentially the same possibilities as the context specification means which are present in the linguistic rules of TooIjP, and thus the same interpretation problems will arise.

The main problem can be reduced to the combination of three operators in one formalism: complementation (the 'not'), alternation (the 'or') and concatenation (the linking of consecutive structures). Therefore, the repetition operator will be omitted in the first part of this study. When a reasonable satisfying solution for the inclusion of complementation has been described, the repetition operator will return in the formalism, together with the simultaneous operator.

The main problem of introducing complementation in regular expressions is that the solution which comes to mind first is not satisfactory. The straightforward approach is to define complementation in a compositional manner, as will be explained in section 3.3. This is done, for instance, by Van der Steen (1987) in the Parspat system. However, for a certain class of expressions the formal interpretation does not correspond to the meaning one expects the expression to have. A simple but not very satisfying solution to the problem is to exclude the cases which cause problems by means of syntactic restrictions, as is done, for instance, in the Fonpars system of Kerkhoff *et al.* (1984). An alternative approach is to study what one expects the patterns of that particular class to mean and try and formalize that expectation so that it can be incorporated in the semantics of the formalism.

In this chapter the latter line of thought is followed. First simplified regular expressions are discussed, i.e., regular expressions stripped of the repetition operator (section 3.2). The formalism does not yet feature a complementation operator, and some properties of this formalism are discussed. Then complementation is included in a straightforward compositional manner, and the cases in which problems arise are identified (section 3.3). The nature of these patterns and the reason why expectation deviates from the formal interpretation are then studied, which leads to a formalization of the expected meaning (section 3.4). This is included in the formalism so that the interpretation of this class of patterns also corresponds to expectation (that is, applying common sense rather than the formal definition to determine its meaning). A discussion on the properties of the thus originated formalism follows (section 3.5). Then, simultaneity and optionality (this is the TooIjP

name for the repetition operator) will be added to the formalism (section 3.6), the consequences of which are also discussed (section 3.7). As will become clear the resulting formalism does not fulfil all the properties one could wish it to have, such as the legitimacy of annihilating double complementation, or applying de Morgan's laws. In adjusting the formalism to practical needs some properties are lost, which are attractive from a theoretical point of view. The discussion (section 3.8) deals with the incompatibility of practical needs and theoretical elegance. Finally, in the concluding section (section 3.9) the most important conclusions are recapitulated.

3.2 Simplified regular expressions

3.2.1 Introduction

The similarity between regular expressions, a target or context pattern in linguistic rules and a search pattern used in a text editor is the fact that *patterns* of characters are specified. These are of exactly the same type as those denoted by regular expressions. To distinguish the formalism here developed from the (standard) regular expressions, the term 'pattern' will be used. In general, a pattern denotes a set of strings, where a string is a sequence of characters.

A typical example of a linguistic rewrite rule, which makes use of such patterns, is given in (3.1):

$$\mathbf{c,h} \rightarrow \mathbf{SJ} \;/\; \_\_ \left\{ \begin{matrix} \mathbf{a,u} \\ \mathbf{vow,q} \end{matrix} \right\} \tag{3.1}$$

The rule states that a character sequence 'ch' should be rewritten into 'SJ' if it is followed by either the character sequence 'au' or a character sequence which is characterized by "a vowel followed by a 'q'". The rule is meant to provide the Dutch pronunciation for the 'ch' in French loans such as 'chauffeur' (driver) or 'choque' (shock). The slash (/) separates the transcription part (left side) from the context specification (right side). The transcription part specifies that the target pattern (**c,h**), which is also called focus, should be rewritten (→) into the change pattern (**SJ**), which is also called the structural change. The comma (,) in the focus denotes the concatenation operator: following a 'c', an 'h' should be present. The context specification specifies in which context the focus should be found in order to apply the transcription. The underscore (__) indicates the position of the focus in the context. Thus in rule (3.1) the left context is empty, which means that any string satisfies the condition. The right context consists of an alternative structure (denoted

by the braces '{' and '}'): of all patterns listed on top of each other (here: **a**, **u** and **vow**, **q**) at least one must be present (here, **vow** is a shorthand notation for the vowels {'a','e','i','o','u'}).

Throughout this chapter, when characters in the input string are referred to, they are placed between quotes (e.g., 'ch'), whereas all patterns are printed in bold face (e.g., **c**, **h**). Also, the patterns will be given in a notation derived from linguistic rules rather than in a notation which is generally used in regular expressions, as the problems were encountered in this specific application.

In rule (3.1) three patterns can be distinguished that are searched for in an input string: the focus, which denotes the string 'ch', the left context, which denotes any arbitrary string, and the right context, which denotes all strings beginning with either the characters 'au', 'aq', 'eq', 'iq', 'oq', or 'uq'. Note the difference between the interpretation of the focus and the contexts. With regard to the focus only strings match which have the correct length and composition, while the right and left contexts are satisfied if the beginning of the strings is correct. For the contexts a 'don't-care pattern', which matches to all strings, is added at the end of the pattern.

A pattern is matched against a string at a certain position, called the *anchor*. Suppose a string consists of 'chauffeur', then matching the right context of (3.1) against the string will only give a positive result if it is started on the third character from the left or, in other words, if the anchor is positioned at the 'a'.

A pattern is also matched in a certain direction. This is determined by the place where the anchor is related to the pattern. For a right context this is at the left side, where the focus (which is the reference point of the linguistic rule) is found. Therefore, the right context is matched from left to right. Conversely, the anchor of the left context is positioned at its right and consequently, the pattern is matched from right to left. It is not so much the direction in which a pattern is matched that is important, as the fact that it can only be done in that specific direction. Consider, for instance, the following right context:

$$\left\{ \begin{array}{c} \mathbf{a} \\ \mathbf{o}, \mathbf{u} \end{array} \right\}, \mathbf{t}$$

This pattern consists of an alternative structure (**a** or **o**, **u**), concatenated by the pattern **t**. The position where a 't' should be found in the string depends on the alternative which is taken into account: if this is the first alternative (**a**), the 't' should be found at the second position to the right of the anchor; for the second alternative (**o**, **u**) this should be the third position.

In the following, I shall only consider left-to-right matching of patterns, as right-to-left matching of reverse patterns will essentially have the same properties. In defining the interpretation of these patterns, I will give the interpretation of a (left or right) context, where, so to speak, the don't-care pattern is added at the end, as that is the most general form of a pattern. The interpretation of a focus pattern can easily be constructed from the general interpretation. Moreover, the don't-care pattern is one of the sources of difficulty and therefore the right context patterns are best suited to illustrate the matter.

3.2.2 The formalism

In example (3.1) quite a few possibilities to construct a pattern are exemplified. First, a pattern may be empty, like the left context. This is denoted by the absence of any pattern. Then, there are the terminals, which refer directly to the character in the input string, for example **c**, **h**, **a** and **u**. These are called *primitives*. Next, one can use a shorthand notation to denote a set of characters, for instance **vow**. Finally, two mechanisms are used to indicate a relation between patterns. These are called *operators*. The comma expresses concatenation: the concatenated patterns should be found successively in the input string. The braces express an 'or' relationship: only one of the specified patterns must be present.

In the formalism presented here only the essential elements which are needed to construct an arbitrary pattern are included, so as to keep the line of argument as clear as possible. Therefore, the possibility for shorthand notation (as in **vow**) is omitted. The complementation operator, used to express a 'not' relationship and which will be denoted by '¬', has been omitted so far and will be introduced presently.

A formalism consists of a syntax and a semantics. The syntax describes which patterns can be constructed. The semantics provides a meaning to those patterns. The syntax and semantics of simplified regular expressions (not featuring complementation) are given below.

Syntax

The syntax of the formalism is given in Table 3.I.

The syntax is given in an informal version of the so-called Backus-Naur Form (Naur, 1963). Here, '::=' defines how a term on the left-hand side can be expanded and '|' denotes alternatives of expansion. In words these rules read:

- A pattern (⟨patt⟩) can be empty ($\emptyset$) or a *non-empty pattern* (⟨ne-patt⟩). In TooJiP the empty pattern is specified by the absence of any pattern.

- A non-empty pattern can be a *structure* (⟨strct⟩) or a *concatenation* (,) of a structure and a non-empty pattern.

Table 3.I: Syntax of simplified regular expressions.

⟨patt⟩ ::= $\emptyset$ | ⟨ne-patt⟩

⟨ne-patt⟩ ::= ⟨strct⟩ | ⟨strct⟩ , ⟨ne-patt⟩

$$\langle \text{strct} \rangle ::= \langle \text{prim} \rangle \mid \left\{ \begin{array}{c} \langle \text{ne-patt} \rangle \\ \langle \text{ne-patt} \rangle \\ \vdots \\ \langle \text{ne-patt} \rangle \end{array} \right\}$$

⟨prim⟩ ::= **a** | **b** | ... | **z**

Table 3.II: Semantics of simplified regular expressions.

$\langle X \rangle = \underline{X}, \underline{\otimes}$

$\underline{\emptyset} = \{\epsilon\}$

$\underline{\otimes} = \{\epsilon, \mathrm{a}, \mathrm{b}, \ldots, \mathrm{z}, \mathrm{aa}, \mathrm{ab}, \ldots, \mathrm{zz}, \mathrm{aaa}, \ldots\} = \sigma^*$

$\underline{x} = \{x\}$ where x is ⟨prim⟩

$$\underline{\left\{ \begin{array}{c} A \\ B \\ \vdots \\ C \end{array} \right\}} = \underline{A} \cup \underline{B} \cup \ldots \cup \underline{C}$$

$\underline{A, B} = \underline{A} \sim \underline{B}$ where A is ⟨strct⟩ and B is ⟨ne-patt⟩

$X \sim Y = \{d_1 d_2 \mid d_1 \in X \wedge d_2 \in Y\}$

- A structure can be a *primitive* (⟨prim⟩) or an *alternative structure*. An alternative structure is denoted by braces ('{' and '}'). It can have any number of arguments and each argument is an arbitrary non-empty pattern (which can be constructed with the rule above).

- A primitive is one of a set of basic symbols, **a**, or **b**, and so on until **z**. Here, only the 26 characters of the alphabet are assumed, but in an actual system like TooIJP other characters such as blanks and interpunction also belong to this set.

The primitives are the building blocks of the formalism. As will follow from the semantics, they refer to information in the input string. The other symbols (such as ',', '{', etc.) are used to define desired relations between the primitives. Examples of patterns are:

$$\mathbf{a},\mathbf{b},\mathbf{c} \quad , \quad \mathbf{c},\left\{\begin{matrix}\mathbf{e}\\ \mathbf{i}\end{matrix}\right\} \quad \text{and} \quad \mathbf{a},\left\{\begin{matrix}\mathbf{b}\\ \mathbf{c},\mathbf{d}\end{matrix}\right\},\mathbf{e} \tag{3.2}$$

Semantics

The semantics of the formalism is given in Table 3.II. They should be read as follows:

- The interpretation of a pattern X (denoted as '$\langle X \rangle$') is the result of a function applied to the concatenation of the pattern X and a special pattern, $\otimes$, the don't-care pattern, to which all strings match. This function is denoted by an underscore and is pronounced as "the meaning of".

- The meaning of the 'empty-pattern', $\emptyset$, is the set containing the empty string. This is a string with length zero and is denoted as 'ϵ'.

- The meaning of the don't-care pattern, $\otimes$, is the set of all strings, where a string is defined as a sequence of arbitrary length composed of arbitrary characters from the alphabet.

- The meaning of a primitive is the set containing the character which is denoted by the primitive. Here, $x = \{x\}$ is shorthand for writing 26 expressions of the type: $\underline{\mathbf{a}} = \{\text{'a'}\}$, $\underline{\mathbf{b}} = \{\text{'b'}\}$, etc.

- The meaning of an alternative structure is the union of the meanings of the individual arguments.

- The meaning of a concatenation of a structure and a non-empty pattern is the *string concatenation* (denoted as '$\sim$') of the meaning of the structure and the meaning of the non-empty pattern (from the syntax it follows that the first term is a structure and the second is a non-empty pattern).
- The string concatenation '$\sim$' of two sets of strings is defined as the set of all strings which can be split up so that the first part (d_1) is an element ($\in$) of the first set (X) and the second part (d_2) is an element of the second set (Y). Note that string concatenation $A \sim B$ is not the same as the carthesian product, $A \times B$. The origin of the substrings cannot be traced in the case of string concatenation: 'att' can be formed by concatenation of 'a' and 'tt' as well as of 'at' and 't'. In carthesian products the origin can be traced: (a,tt) is considered to be a different pair from (at,t).

The function "the meaning of", in fact, closely resembles the definition of regular expressions. It deviates at only two small points. One is that $\emptyset$ denotes the empty string 'ϵ' rather than the empty set. The second point is that the repetition operator is absent. This last point, however, is not essential, and it will be re-included in section 3.6. This formalism, based on the regular expressions, but stripped of the repetition operator will be called 'simplified regular expressions'.

According to this semantics, the interpretation of the examples given in (3.2) is respectively: "the set of all strings beginning with 'abc'", "the set of all strings beginning with 'ce' or 'ci'" and "the set of strings beginning with 'abe' or 'acde'".

Some properties of the formalism

An important property of a formalism is that it is compositional. Compositionality means that the meaning of a pattern can be expressed in terms of the meanings of its composing patterns. If a formalism is compositional, we have the guarantee that we can always determine the meaning of a pattern, irrespective of its complexity. As can be seen in Table 3.II, the semantics fulfils these compositionality requirements, so the formalism is compositional.

Another property of the present formalism is the distributivity of the alternation over concatenation. For instance, one would expect that relation (3.3) holds:

$$\left\{ \begin{matrix} \mathbf{a} \\ \mathbf{b,c} \end{matrix} \right\} \mathbf{,d} = \left\{ \begin{matrix} \mathbf{a,d} \\ \mathbf{b,c,d} \end{matrix} \right\} \tag{3.3}$$

In other words, it is permitted to put freely an arbitrary pattern within braces, or take equal patterns out. Given the definition of string concatenation (the last line of Table 3.II), one can prove (see Appendix 3.A) that one may do so.

A final important property of the present formalism is that it corresponds to one's intuition, it does what one expects it to do. Without having to apply step by step the rules of Table 3.II, the meaning of a pattern is 'obvious'. In my opinion, this is an essential property of a formalism, as otherwise it would never be satisfactory in practice.

3.3 Complementation introduced in a compositional manner

The user's possibilities for constructing patterns with these simplified regular expressions can be summarized as follows. With the primitives one can impose the basic restrictions on the input: in order to have the input match the pattern, the input character against which the primitive is matched should meet the requirements imposed by that primitive[1]. With the concatenation operator one can specify the desired presence of a string, i.e., the respective requirements of the primitives should be met by the respective characters of the input string. With the alternative operator one can express an 'or' relation: at the appropriate position in the input string one of the patterns specified in the alternative structure should be present.

The question arose as to whether other operators could also be introduced in the formalism. Regular expressions can be expected to be powerful enough to specify the patterns that one needs, so it is not so much the power of expression one wants to increase, as the ease of expression. With an analogon of the 'and' and the 'not' certain patterns can be expressed more elegantly and transparently than by means of enumeration using alternatives. Also an optional or repetition operator can be useful for certain expressions.

We will first consider the introduction of only the 'not', since the main problem of including additional operators lies here. The introduction of the other operators will be discussed in section 3.6. The 'not', called the *complementation operator*, is denoted by '¬'. For instance, '¬c' should be interpreted as "any character except 'c'". Since a pattern denotes a set of strings, an inverted pattern will also denote a set of strings. In general, one cannot negate a set, but one must take the complement of a set. This involves selecting all elements from a *universe*, excluding the elements of the set. For this reason the operator is called a complementation operator, rather than a negation or inversion operator.

[1] In this chapter, the primitives only consist of the characters of the alphabet, but as explained in chapter 2, a primitive can also denote a set of characters, such as the vowels.

The most straightforward interpretation of a complemented pattern is the complement of the set the pattern denotes. This is expressed mathematically in (3.4):

$$\underline{\neg X} = U \setminus \underline{X} \tag{3.4}$$

Here, U is the universe relative to which the complementation operates, and '\' is the operator of set-difference, which in (3.4) subtracts $\underline{X}$ from U. However, this universe U must still be defined.

The first possibility that comes to mind is the universe which is generally used with regular expressions, the set of all strings from a given alphabet. This is often denoted as σ^* (σ is the alphabet, * is the repetition operator). This choice of universe, however, has a serious drawback. Consider the pattern '¬**c**'. According to definition (3.4) a single 'c' is excluded from the set of strings denoted by the pattern, as expected. The string 'cc', however, is not excluded from this set, nor is 'ca', nor any other string of at least two characters, irrespective of whether it starts with a 'c' or not. And since in the application of context specification the context generally consists of more than one character, the use of complementation with $U = \sigma^*$ does not seem very useful.

The pattern '¬**c**' creates the impression that the first character to be tested may not be a 'c', but further (on the following characters) no restrictions are made, so '¬**c**' would denote all strings not starting with a 'c'. In the same way, '¬[**c**]**,t**' creates the impression of denoting all strings not starting with a 'c' and having a 't' in second position.

The phrase "creates the impression" is a central notion in the argument. Of course, one can introduce any operator in any formalism and attach it to any definition, but this is only useful if it works out according to one's expectation or intuition, or at least does not work out counter-intuitively. And since the pattern '¬**c**' does not mean (according to (3.4)) the same as the impression it creates, I do not consider this definition to be useful.

This can be fixed by adjusting the definition of the universe. It seems that when complementation is used, the universe with respect to which it operates is given implicitly. '¬**c**' means any character but 'c', '¬[**c**,**t**]' means any two characters but the sequence 'ct'. Therefore, it seems satisfactory to make a character count in the complemented pattern: the number of concatenated primitives determines the length and all strings of that length are included in the universe. Thus, if a single primitive is complemented, the universe is the set of strings with length one; if a concatenation of three primitives is complemented, the universe is the set of strings with length three; if an alternative structure is complemented, that for instance contains two *paths* of

different length, such as in $\neg\{^{\mathbf{a}}_{\mathbf{o,u}}\}$, the union of the universes of the individual patterns is taken, thus in this case all strings of length one and two. This is formalized in a new definition, (3.5):

$$\underline{\neg X} = X^U \setminus \underline{X} \tag{3.5}$$

Note that the universe with respect to which the complementation operates is now being determined explicitly by the pattern which is being complemented.

The new interpretation of the universe of an arbitrary pattern is formalized in Table 3.III. The universe of a primitive is the alphabet. The universe of a concatenation is the string concatenation of the universes of the composing patterns. This provides the 'character count'. The alternative operator unites the universes of the composing patterns. Finally, the universe of a complemented pattern is the same as the universe of the non-complemented pattern.

The possibilities to construct a pattern can now be extended. In Table 3.IV a new syntax is given to replace the former one (given in Table 3.I). The complementation operator is included as a structure; everywhere where one can use a primitive or an alternative operator, one can now also use the complementation operator to complement an arbitrary non-empty pattern. If the complemented structure consists of a single structure one may omit the square brackets which serve to denote the range of the complementation[2].

[2]Square brackets are chosen since parentheses will be used for optionality. Although square bracket will also be used for another purpose, viz. to denote simultaneity, there is no ambiguity since simultaneity always has two or more arguments, whereas optionality only has one.

Table 3.III: Definition of the universe of an extended regular expression.

$x^U = \{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{z}\} = \sigma$	where x is ⟨prim⟩
$(A, B)^U = A^U \sim B^U$	where A is ⟨strct⟩ and B is ⟨ne-patt⟩
$(\left\{\begin{matrix} A \\ B \\ \vdots \\ C \end{matrix}\right\})^U = A^U \cup B^U \cup \ldots \cup C^U$	
$(\neg A)^U = A^U$	

Table 3.IV: The syntax of extended regular expressions.

$$\langle \text{patt} \rangle ::= \emptyset \mid \langle \text{ne-patt} \rangle$$

$$\langle \text{ne-patt} \rangle ::= \langle \text{strct} \rangle \mid \langle \text{strct} \rangle \, , \langle \text{ne-patt} \rangle$$

$$\langle \text{strct} \rangle ::= \langle \text{prim} \rangle \mid \left\{ \begin{array}{c} \langle \text{ne-patt} \rangle \\ \langle \text{ne-patt} \rangle \\ \vdots \\ \langle \text{ne-patt} \rangle \end{array} \right\} \mid \neg[\langle \text{ne-patt} \rangle] \mid \neg\langle \text{strct} \rangle$$

$$\langle \text{prim} \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \ldots \mid \mathbf{z}$$

Table 3.V: The semantics of regular expressions extended with complementation in a compositional manner.

$$\langle X \rangle = \underline{X \, , \otimes}$$

$$\underline{\emptyset} = \{\epsilon\}$$

$$\underline{\otimes} = \sigma^*$$

$$\underline{x} = \{x\} \qquad \text{where } x \text{ is } \langle \text{prim} \rangle$$

$$\underline{\left\{ \begin{array}{c} A \\ B \\ \vdots \\ C \end{array} \right\}} = \underline{A} \cup \underline{B} \cup \ldots \cup \underline{C}$$

$$\underline{\neg[A]} = A^U \setminus \underline{A}$$

$$\underline{A \, , B} = \underline{A} \sim \underline{B} \qquad \text{where } A \text{ is } \langle \text{strct} \rangle \text{ and } B \text{ is } \langle \text{ne-patt} \rangle$$

$$X \sim Y = \{d_1 d_2 \mid d_1 \in X \wedge d_2 \in Y\}$$

The semantics accompanying this syntax is given in Table 3.V and replaces the former semantics (given in Table 3.II). Likewise, the new semantics differs only from the old one in the respect that the interpretation for the complementation operator as given in (3.5) is included. The Parspat system described by Van der Steen (1987) features complementation exactly in accordance with this scheme.

3.3.1 Some examples

With the new formalism defined by Tables 3.III, 3.IV and 3.V, we can determine the interpretation of the patterns which we can construct with the new complementation operator. Starting with simple patterns, the interpretation of some of these is given according to the new semantics.

$$\langle \neg \mathbf{c} \rangle = (\sigma \setminus \{\mathrm{c}\}) \sim \underline{\otimes} = \{\mathrm{a}, \mathrm{b}, \mathrm{d}, \ldots, \mathrm{z}\} \sim \sigma^* \tag{3.6}$$

In words, pattern (3.6) is satisfied by an arbitrary non-empty string, which does not start with a 'c'. This is, indeed, the set we previously described informally. The other two patterns we saw earlier behave in accordance with expectation, too.

$$\langle \neg \mathbf{c}\,, \mathbf{t} \rangle = \{\mathrm{a}, \mathrm{b}, \mathrm{d}, \ldots, \mathrm{z}\} \sim \mathrm{t} \sim \sigma^* \tag{3.7}$$

Here, any non-'c' character must be followed by a 't'. Note that the range of the complementation is restricted to the **c**; the **t** is a 'positive' pattern that must be present in the input string. To include the **t** in the complementation, one should specify:

$$\langle \neg [\mathbf{c}\,, \mathbf{t}] \rangle = \{\mathrm{aa}, \mathrm{ab}, \ldots, \mathrm{cs}, \mathrm{cu}, \ldots, \mathrm{zz}\} \sim \sigma^* \tag{3.8}$$

Now, the string may start with any two characters, 'ct' excluded.

Another pattern, for instance, is (3.9), where the use of the alternative operator in combination with the complementation operator is illustrated.

$$\left\langle \neg \left\{ \begin{matrix} \mathbf{a} \\ \mathbf{c} \end{matrix} \right\}, \mathbf{t} \right\rangle = \{\mathrm{bt}, \mathrm{dt}, \ldots, \mathrm{zt}\} \sim \sigma^* \tag{3.9}$$

Note that both the 'a' and the 'c' are excluded as first character. The second character, as expected, must be a 't'.

3.3.2 Some problem cases

So far, the interpretation of these patterns corresponds to our intuition of what they should be; complementing patterns lead to exclusion of the specified characters, the other characters being included. However, we can also construct other more complex patterns, in which case complications arise. Consider for instance the following pattern:

$$\left\langle \neg \left\{ \begin{matrix} \mathbf{a} \\ \mathbf{o}, \mathbf{u} \end{matrix} \right\}, \mathbf{t} \right\rangle = \{\mathrm{bt}, \mathrm{ct}, \ldots, \mathrm{zt}, \mathrm{aat}, \ldots, \mathrm{att}, \ldots, \mathrm{ott}, \mathrm{ovt}, \ldots \mathrm{zzt}\} \sim \sigma^* \tag{3.10}$$

This pattern includes some unexpected strings. As can be seen, the strings 'att...' (i.e., all strings starting with the characters 'att') are approved of, due to the $\neg[\mathbf{o}, \mathbf{u}], \mathbf{t}$ path. This path introduces all two-character sequences except 'ou'—which includes 'at'—to be concatenated to 't...'. This is undesirable, as the other path of (3.10), $\neg(\mathbf{a}), \mathbf{t}$ is meant to exclude the strings starting with 'at', of which the strings 'att...' are a subset.

Pattern (3.11) is even worse.

$$\left\langle \neg \left\{ \begin{matrix} \mathbf{a} \\ \mathbf{o}, \mathbf{u} \end{matrix} \right\} \right\rangle = \{\mathrm{b}, \mathrm{c}, \ldots, \mathrm{o}, \ldots, \mathrm{z}, \mathrm{aa}, \ldots, \mathrm{az}, \mathrm{ba}, \ldots, \mathrm{ot}, \mathrm{ov}, \ldots \mathrm{zz}\} \sim \sigma^* = \sigma^* \tag{3.11}$$

First, it includes all strings starting with an 'a', as $\neg[\mathbf{o}, \mathbf{u}]$ approves of all two-character sequences starting with 'a'. But also the strings starting with 'ou' are included: the 'o' is approved by $\neg\mathbf{a}$, to be string concatenated with $\otimes$. Amongst other strings, the strings 'u...' are linked to the 'o'. Thus it is found that (3.11) is a clumsy way to specify the alphabet, σ^*. This is clearly unexpected as the pattern suggests that all strings starting with 'a' or 'ou' are to be excluded.

These are two sample patterns of a class for which there is a clash between the formal interpretation of the pattern and our expectation of its meaning. Although the patterns may be of a somewhat hypothetical nature, patterns of a similar construction might well be specified in the application for which the formalism is devised. In natural languages one can distinguish clusters of letters which share a certain property. These letter clusters do not necessarily have the same length. The class of English vowels, for instance, can consist of one character, as the 'o' in 'over', but also of two, as 'ea' in 'reach'. Another example is the class of so-called well-formed initial consonant clusters. These

are sequences of consonants one may encounter at the beginning of a syllable. For instance, 'str' is encountered in words such as 'string' and 'strange', while 'kp' will never form the beginning of a syllable in English. One may want to trigger a rule on the basis of such a class, or exclude the application of a rule in such circumstances. The complementation operator is well suited for this purpose, and typically, when excluding such classes this leads to patterns of the above type.

Since linguists in general cannot be assumed to be familiar with the theory of extended regular expressions, it is a drawback that patterns like (3.10) and (3.11) do not behave as one expects. Therefore, the semantics of Table 3.V is not completely satisfactory.

In the next section we will try to find a solution to this problem. What we expect a pattern to denote, however, is an intuitive notion, which is difficult to work with if it cannot be made explicit. Therefore, we will try to find the type of patterns for which the expected and formal interpretation diverge, analyse the expected meaning of these patterns and formalize the expected interpretation.

3.4 Explicit nofits

3.4.1 Succeeding structure

Before we start to try and make this intuitive expectation explicit, we take another look at the patterns (3.6)–(3.11). It is found that the divergence of expectation and formal interpretation only occurs when one complements a pattern that denotes strings of different length. If this is the case, as in (3.10) and (3.11), a string which is matched against that pattern can be split up in more than one way. Due to these different divisions, it can occur that strings which are rejected by one path (for instance, the strings 'att...' are among the strings which are rejected by $\neg[\mathbf{a}],\mathbf{t}$ in (3.10)) are approved by another ('att...' satisfies $\neg[\mathbf{o},\mathbf{u}],\mathbf{t}$).

Clearly, these effects are not intended when such patterns are specified. If we take another look at pattern (3.12):

$$\neg\left\{\begin{array}{c}\mathbf{a}\\ \mathbf{o},\mathbf{u}\end{array}\right\},\mathbf{t} \tag{3.12}$$

one expects it to mean something like: "In the input string, somewhere a 't' must be found, but preceding the 't' a single 'a' and the sequence 'ou' are not allowed." Informally speaking, it is as if the input string is fitted on

the pattern, so that for a certain path in the pattern each primitive will be matched to one segment. If the string fits in such a way that its segments match both the complemented and the non-complemented parts of a pattern (as 'att...' does for $\neg[\mathbf{a}]\,,\mathbf{t}$) the string is explicitly meant not to match. Such strings will be called *explicit nofits*.

On the other hand, strings which do not belong to that category and have a *matching condition*, should be included. A matching condition occurs if the string can be fitted to the pattern in such a way that the segments fitted on the complemented parts of the pattern do *not* match and the segments fitted on the non-complemented (positive) parts *do*. Strings which fulfil the matching condition are called *candidates*. For instance, both the strings 'att...' and 'art...' are matching strings of (3.12) as they fulfil the matching condition, viz. on the path $\neg[\mathbf{o}\,,\mathbf{u}]\,,\mathbf{t}$. However, as 'att...' at the same time is also an explicit nofit, viz. on the path $\neg\mathbf{a}\,,\mathbf{t}$, it should not be included in the set of strings which are denoted by (3.12). The string 'art...', on the other hand, is not an explicit nofit and is therefore included in the set.

This informal description of expectation can be formalized and expressed mathematically:

$$\textit{explicit nofits}\ (\neg A\,,B) \;=\; \underline{A}{\sim}\underline{B} \tag{3.13}$$

$$\textit{candidates}\ (\neg A\,,B) \;=\; (A^{U}\backslash\,\underline{A}){\sim}\underline{B} \tag{3.14}$$

In the compositional semantics of Table 3.V the complementation operator selects the candidates, without excluding the explicit nofits. To exclude these, a new definition for complementation is proposed: the set of candidates minus the set of explicit nofits, which is expressed mathematically in (3.15):

$$\underline{\neg A\,,B} = (A^{U}\backslash\,\underline{A}){\sim}\underline{B}\,\backslash\,\underline{A}{\sim}\underline{B} \tag{3.15}$$

String-concatenation ($\sim$) has higher precedence than set-difference ($\backslash$). As is proved in Appendix 3.B, (3.15) can be simplified to (3.16):

$$\underline{\neg A\,,B} = A^{U}{\sim}\underline{B}\,\backslash\,\underline{A}{\sim}\underline{B} \tag{3.16}$$

One can comprehend this as follows: the strings which are added to the generating set (the set to the left of the set-difference sign '$\backslash$') of (3.16) as compared with (3.15) are all part of the set of explicit nofits which are excluded afterwards, so the resulting set of strings are the same for (3.15) and (3.16).

This way of presenting the complementation expresses the fact that the complementation is sensitive to the succeeding pattern. If we see the complementation as an operator which operates on one pattern, in (3.16) we lose the strict compositional character that the semantics of Table 3.V had, as we include the succeeding pattern in the definition of complementation.

3.4.2 Closing brackets

Before we can include (3.16) in a new proposal for the semantics, there is another observation to be made. In (3.3) we saw that we can put a pattern succeeding an alternative structure into the structure. Now suppose we have a pattern resembling (3.3), in which complementation is used:

$$\left\{ \begin{matrix} \mathbf{a} \\ \mathbf{b}, \neg\mathbf{c} \end{matrix} \right\}, \mathbf{d} \tag{3.17}$$

Again, we can identify two paths, one stating $\mathbf{a}, \mathbf{d}$ and the other stating $\mathbf{b}, \neg\mathbf{c}, \mathbf{d}$. In the second path, $\mathbf{d}$ is the pattern succeeding the complementation $\neg\mathbf{c}$. However, we cannot simply apply the semantics of Table 3.V with the new definition of complementation. The complementation will not "see" the succeeding pattern $\mathbf{d}$, as it is outside of the alternative structure:

$$\left\langle \left\{ \begin{matrix} \mathbf{a} \\ \mathbf{b}, \neg\mathbf{c} \end{matrix} \right\}, \mathbf{d} \right\rangle = \underline{\left\{ \begin{matrix} \mathbf{a} \\ \mathbf{b}, \neg\mathbf{c} \end{matrix} \right\}, \mathbf{d}, \otimes} = \underline{\left\{ \begin{matrix} \mathbf{a} \\ \mathbf{b}, \neg\mathbf{c} \end{matrix} \right\} \sim \mathbf{d}, \otimes} \tag{3.18}$$

Moreover, it is found in this case that there is no pattern present at all behind the $\neg\mathbf{c}$ in (3.17), to substitute for pattern B of (3.16).

Both objections can be resolved by putting patterns which succeed an alternative structure into braces, for instance, by reformulating (3.17) into (3.19):

$$\left\{ \begin{matrix} \mathbf{a}, \mathbf{d} \\ \mathbf{b}, \neg\mathbf{c}, \mathbf{d} \end{matrix} \right\} \tag{3.19}$$

In this way, we establish that

(a) a succeeding pattern (pattern B of (3.16)) is always present, and
(b) the full succeeding pattern is within the scope of the complementation.

Note that if the original pattern does not have a succeeding pattern, as in (3.11), the formalism adds one, the $\otimes$-pattern. This has the desired effect, as for instance in (3.11) all strings 'a...' and 'ou...' are included in the set of explicit nofits, and thus are excluded from the new interpretation of (3.11).

3.4.3 A semantics excluding explicit nofits

Of course, we do not want to burden the user with the process of putting the succeeding pattern into the scope of a structure, and moreover we still

want the property of being able to manoeuvre patterns freely in and outside brackets to be preserved. Therefore, the formalism must provide a mechanism to put these patterns into brackets.

This can be achieved by approaching the patterns from their concatenation aspect. As can be seen in Table 3.V, the first line of the semantics concatenates the ⊗-pattern to an arbitrary pattern before the function "the meaning of" is applied. This means that for any user-specified structure there will always be a succeeding pattern: either a user-defined (non-empty) pattern or the ⊗-pattern. Then, for alternative structures, the succeeding pattern can be brought into its scope:

$$\underline{\left\{\begin{matrix} A \\ B \\ \vdots \\ C \end{matrix}\right\}, Y} = \underline{\left\{\begin{matrix} A, Y \\ B, Y \\ \vdots \\ C, Y \end{matrix}\right\}} = \underline{A, Y} \cup \underline{B, Y} \cup \ldots \cup \underline{C, Y} \tag{3.20}$$

Note that, just like (3.16), the meaning of this structure is now defined in combination with its succeeding structure.

To specify the meaning of a concatenation in general, we must discriminate between the various leftmost structures. For concatenation and alternation the meaning is now defined: (3.16) and (3.20). Two more possibilities remain for the leftmost structure, the empty pattern and the primitives (see Table 3.IV). These are directly derived from Table 3.V. As a whole this results in Table 3.VI, where the new semantics for the formalism is proposed.

There remain some minor points to discuss, due to the definition of complementation. In determining the meaning of the complementation, the meaning of the complemented structure ($\underline{A}$) has to be determined, and eventually we are left with the task of determining the meaning of a single structure (i.e., which does not contain a concatenation). This can be any of the three structures: a primitive, an alternation or a complementation. To avoid the necessity of specifying the meaning of these structures, which essentially have the same characteristics as those already specified for concatenation, the empty pattern '$\emptyset$' is concatenated to that structure ($\underline{X} = \underline{X, \emptyset}$). This does not alter the interpretation of the pattern, as string concatenation of the empty string to an arbitrary string does not alter the string. We can once again apply the rule for the meaning of a concatenation. Eventually, we will only have to determine the meaning of the empty pattern, which, for this reason, is included in the semantics. Thus the semantics of Table 3.VI is complete; the meaning of any arbitrary pattern can be determined.

3.5 Properties of the new semantics

3.5.1 Consistent versus inconsistent patterns

The first thing to check is whether the new formalism yields the same results for those patterns which in the old interpretation already had the desired meaning. For this purpose, it is necessary to reconsider the difference between the patterns in section 3.3.1 and those in section 3.3.2. The interpretation problems appear when it is possible to find a string which is divisible in different ways, so that it is rejected by one path and approved of by another (see section 3.4.1), or, in other words, the string is both a candidate and an explicit nofit. Such strings are called *inconsistent divisible strings*. Patterns for which such strings exist are called, in imitation, *inconsistent patterns*.

Table 3.VI: The semantics of regular expressions extended with complementation in a semi-compositional manner.

$$\langle X \rangle = \underline{X}, \underline{\otimes}$$

$$\underline{\emptyset} = \{\epsilon\}$$

$$\underline{\otimes} = \sigma^*$$

$$\begin{array}{lllll} \underline{X,Y} = & \text{if} & X = \emptyset & \text{then} & \underline{Y} \\ & \text{if} & X = x & \text{then} & \{x\}{\sim}\underline{Y} \quad \text{where } x \text{ is } \langle \text{prim} \rangle \\ & \text{if} & X = \left\{ \begin{array}{c} A \\ B \\ \vdots \\ C \end{array} \right\} & \text{then} & \underline{A,Y} \cup \underline{B,Y} \cup \ldots \cup \underline{C,Y} \\ & \text{if} & X = \neg A & \text{then} & A^{U}{\sim}\underline{Y} \setminus \underline{A}{\sim}\underline{Y} \end{array}$$

$$\underline{X} = \underline{X,\emptyset} \qquad \text{where } X \text{ is } \langle \text{strct} \rangle$$

$$X{\sim}Y = \{d_1 d_2 \mid d_1 \in X \wedge d_2 \in Y\}$$

Consequently, all patterns which are not inconsistent are consistent patterns[3].

Defined semantically as such, there is no straightforward characteristic of appearance to distinguish consistent patterns from inconsistent ones. For instance,

$$\neg\left\{\begin{matrix}\mathbf{a}\\ \mathbf{o},\mathbf{u}\end{matrix}\right\},\mathbf{t}$$

is an inconsistent pattern, as 'att...' is both a candidate and an explicit nofit. On the other hand, a pattern which closely resembles the previous one,

$$\neg\left\{\begin{matrix}\mathbf{a}\\ \mathbf{o},\mathbf{u}\end{matrix}\right\},\mathbf{t},\mathbf{r}$$

is consistent, as no inconsistent divisible strings exists.

However, there is a characteristic of appearance which guarantees consistency. One can prove (see Appendix 3.C) that patterns containing only positive structures are consistent. Also, if a pattern contains complemented structures and the complementation contains only paths of a certain, specific length, this pattern is consistent, too. So, for instance,

$$\neg\left\{\begin{matrix}\mathbf{a}\\ \mathbf{e}\end{matrix}\right\},\mathbf{t},\neg\left\{\begin{matrix}\mathbf{o},\mathbf{u}\\ \mathbf{e},\mathbf{a}\end{matrix}\right\},\mathbf{r}$$

is a consistent pattern as it fulfils the characteristics of appearance.

3.5.2 Relation between the two definitions of the semantics

The inconsistent divisible strings for a certain pattern are those strings which are both a candidate (the approving path) and an explicit nofit (the rejecting path). For consistent patterns there are no such strings, which means that a string is exclusively a candidate, an explicit nofit, or neither.

Given the presumption that a pattern is consistent it is proved in Appendix 3.C that the old semantics of Table 3.V yield the same results as the new semantics of Table 3.VI. An illustration of this fact is that the only essential difference between the two semantics, which lies in the definition of complementation, disappears for consistent patterns. This can be seen as follows. For consistent patterns there is no overlap between candidates and explicit nofits. Mathematically expressed this means that $(A^U \setminus \underline{A}) \sim \underline{B} \cap \underline{A} \sim \underline{B} = \emptyset$.

[3] A formal definition is given in Appendix 3.C.

Then $\left[(A^U \setminus \underline{A}) \sim \underline{B}\right] = \left[(A^U \setminus \underline{A}) \sim \underline{B}\right] \setminus \underline{A} \sim \underline{B}$. The left-hand side of this equation is the old definition, and the right-hand side is equal to the new definition (see the simplification of (3.15) to (3.16)).

For inconsistent patterns, on the other hand, the two semantics yield of course a different result. This is to be expected, as for these patterns the old semantics yield undesirable results, which is resolved in the new semantics.

3.5.3 Double complementation

The new formalism lacks, however, a property which one could also expect a formalism featuring complementation to have: the possibility to annihilate double complementation, i.e., a complemented structure which is directly being complemented. In set theory, for instance, this is a characteristic property which holds. The fact that this property does not hold for the proposed formalism can be seen as follows:

$$\underline{\neg\neg A\,,B} = (\neg A)^U \sim \underline{B} \setminus \underline{\neg A} \sim \underline{B} = A^U \sim \underline{B} \setminus (A^U \setminus \underline{A}) \sim \underline{B} \tag{3.21}$$

For inconsistent patterns inconsistent divisible strings exist, which means that they are candidates as well as explicit nofits. In (3.21) one can see that such strings will be excluded from the double complementation $\neg\neg A\,,B$, as they are candidates to $\neg A\,,B$. At the same time, however, those strings are explicit nofits to $\neg A\,,B$. If, for instance, A itself does not contain another complementation, the explicit nofits to $\neg A\,,B$ will match $A\,,B$. Thus, such strings match $A\,,B$, but do not match $\neg\neg A\,,B$. For instance:

$$\mathrm{att}\sigma^* \in \left\langle \left\{ \begin{matrix} \mathbf{a} \\ \mathbf{o}\,,\mathbf{u} \end{matrix} \right\}, \mathbf{t} \right\rangle \quad \text{but} \quad \mathrm{att}\sigma^* \notin \left\langle \neg\neg \left\{ \begin{matrix} \mathbf{a} \\ \mathbf{o}\,,\mathbf{u} \end{matrix} \right\}, \mathbf{t} \right\rangle \tag{3.22}$$

Attempts to adjust the formalism so that both the existing properties are retained and double complementation can be annihilated all fail for one reason or the other. Appendix 3.D elaborates on this matter. Two alternative definitions for complementation are discussed, in which essentially only the definition of explicit nofits is adjusted slightly. It appears that if double complementation may be annihilated, the formal interpretation of some patterns differs from what one would expect, and, vice versa, fixing the interpretation to expectation once again deletes the legitimacy of annihilating double complementation.

3.5.4 Power of expression

The introduction of complementation as an operator in simplified regular expressions does not increase the power of expression of the formalism. This can be seen as follows:

$$\underline{\neg A\,,B} = A^{U}{\sim}\underline{B} \setminus \underline{A}{\sim}\underline{B} = A^{U}{\sim}\underline{B} \cap \overline{\underline{A}{\sim}\underline{B}} \tag{3.23}$$

Here, the overbar denotes complementation with respect to σ^*, the universe generally used for complementing regular expressions as a whole. Since regular sets are closed under complementation (in the regular sense) and under intersection, the complementation operator as it is introduced here does not increase the power of expression, or, in other words: one can always devise a pattern without using the complementation operator which denotes the same set as the one denoted by a pattern in which the complementation operator is used.

However, the complementation operator was not introduced to increase a user's power of expression. Rather, the aim was to increase the ease of expression. With the complementation operator exceptions to rules can be stated more elegantly in an explicit manner, which results in both more concise and more transparent rules.

3.6 Including simultaneity and optionality

Now that complementation has been included in the semantics in a reasonably satisfying manner, we can study the consequences of including two more operators. This concerns simultaneity, the 'and' operator, and optionality, which can be used for optional and repetitive structures. The operators will be introduced in the philosophy of excluding explicit nofits, which means that these operators, too, include the succeeding pattern into their scope. With respect to the interpretation of the simultaneous operator this gives rise to some minor additional complications, which are discussed subsequently. First, therefore, the optional operator is dealt with, then the simultaneous operator is discussed.

3.6.1 The optional operator

The introduction of the optional operator is fairly straightforward. With some minor exceptions the optional operator is just a shorthand for a specific type of alternative structure. The general format is as follows:

$(A)a\text{-}b$

The parentheses denote the optional structure, A is a non-empty pattern, and a and b denote respectively the minimum number of times that A must, and the maximum number it may, be present. Therefore, it is required that $a \leq b$. If 'b' is omitted the pattern is interpreted as a or more times pattern A. If both a and b are omitted the pattern is interpreted as 'optionally A', i.e., zero or one A, which explains the name of this operator. Since in TooIiP this is the main application of this operator, the operator is called 'optional' rather than 'repetitive'. Examples of its use are given in section 2.3.

Since optionality is merely a shorthand for alternatives, it can be included in the formalism in accordance with the alternative operator. The relevant definitions are given below:

$$\begin{aligned}
\underline{X,Y} = \quad & \text{if} \quad X = (A)a\text{-}b \quad \text{then} \quad \underline{A^a,Y} \cup \underline{A^{a+1},Y} \cup \ldots \cup \underline{A^b,Y} \\
& \text{if} \quad X = (A)a \quad \text{then} \quad \underline{A^a,Y} \cup \underline{A^{a+1},Y} \cup \ldots \\
& \text{if} \quad X = (A) \quad \text{then} \quad \underline{Y} \cup \underline{A,Y}
\end{aligned}$$

Here, $A^0 = \emptyset$ and $A^i = A, A^{i-1}$. A^0 is defined as $\emptyset$ so as to let the zero repetition denote the empty string.

Since each structure can be used in any place, the optional structure can also be complemented. The function of determining the universe must therefore also be capable to determine the universe of the optional structure. Since the universe serves as a character counting mechanism, the universe for optional structures is defined as follows:

$$\begin{aligned}
&\left((A)a\text{-}b\right)^U = \left(A^a\right)^U \cup \left(A^{a+1}\right)^U \cup \ldots \cup \left(A^b\right)^U \\
&\left((A)a\right)^U = \left(A^a\right)^U \cup \left(A^{a+1}\right)^U \cup \ldots \\
&\left((A)\right)^U = \{\epsilon\} \cup A^U \\
&\emptyset^U = \epsilon
\end{aligned}$$

The clause $\emptyset^U = \epsilon$ is included for the zero repetition case.

3.6.2 The simultaneous operator

The simultaneous operator can be introduced analogously to the alternative operator. Thus, the general format is as follows:

$$\begin{bmatrix} A \\ B \\ \vdots \\ C \end{bmatrix}$$

All arguments are non-empty patterns. The simultaneous structure can have 2 or more arguments. The intuitive meaning of this structure is that where it is specified the input string should meet the requirements of all arguments. Retaining the symmetry with alternation, simultaneity is defined as follows:

$$\underline{X,Y} = \text{ if } X = \begin{bmatrix} A \\ B \\ \vdots \\ C \end{bmatrix} \text{ then } \underline{A,Y} \cap \underline{B,Y} \cap \ldots \cap \underline{C,Y} \qquad (3.24)$$

For the simultaneous operator, too, the universe must be defined. In the light of the character count it is defined as follows:

$$(\begin{bmatrix} A \\ B \\ \vdots \\ C \end{bmatrix})^U = A^U \cup B^U \cup \ldots \cup C^U$$

Combining these definitions with the current formalism, the complete formalism which defines the interpretation of patterns is given by tables 3.VII, 3.VIII and 3.IX. For reasons which will become clear this formalism is called the semi-compositional formalism for extended regular expressions.

3.7 Properties of the semi-compositional formalism

3.7.1 Explicit nofits

Concatenating the succeeding structure to the individual arguments of the simultaneous structure fits in the philosophy of excluding explicit nofits. Consider, for instance, pattern (3.25):

$$\begin{bmatrix} \textbf{(cons)1-2} \\ \neg\{\begin{matrix} \textbf{b} \\ \textbf{c,d} \end{matrix}\} \end{bmatrix}\textbf{,f} \qquad (3.25)$$

Here, **cons** is a macro which denotes all consonantal segments, which includes, for instance, 'b', 'c', 'd' and 'f'.

Table 3.VII: The semantics of patterns.

Semantics:

$$\langle X \rangle = \underline{X, \otimes}$$

$$\underline{\emptyset} = \{\epsilon\}$$

$$\underline{\otimes} = \sigma^*$$

$$\begin{array}{lllll}
\underline{X,Y} = & \text{if} & X = \emptyset & \text{then} & \underline{Y} \\
& \text{if} & X = x & \text{then} & \{x\}{\sim}\underline{Y} \quad \text{where } x \text{ is } \langle \text{prim} \rangle \\
& \text{if} & X = \left\{ \begin{array}{c} A \\ B \\ \vdots \\ C \end{array} \right\} & \text{then} & \underline{A,Y} \cup \underline{B,Y} \cup \ldots \cup \underline{C,Y} \\
& \text{if} & X = \left[\begin{array}{c} A \\ B \\ \vdots \\ C \end{array} \right] & \text{then} & \underline{A,Y} \cap \underline{B,Y} \cap \ldots \cap \underline{C,Y} \\
& \text{if} & X = \neg A & \text{then} & A^U{\sim}\underline{Y} \setminus \underline{A}{\sim}\underline{Y} \\
& \text{if} & X = (A)a\text{-}b & \text{then} & \underline{A^a,Y} \cup \underline{A^{a+1},Y} \cup \ldots \cup \underline{A^b,Y} \\
& \text{if} & X = (A)a & \text{then} & \underline{A^a,Y} \cup \underline{A^{a+1},Y} \cup \ldots \\
& \text{if} & X = (A) & \text{then} & \underline{Y} \cup \underline{A,Y}
\end{array}$$

$$\underline{X} = \underline{X,\emptyset} \qquad \text{where } X \text{ is } \langle \text{strct} \rangle$$

$$X{\sim}Y = \{d_1 d_2 \mid d_1 \in X \wedge d_2 \in Y\}$$
$$A^0 = \emptyset$$
$$A^i = A, A^{i-1} \qquad i \geq 1$$

Table 3.VIII: The syntax of patterns.

Syntax:

$$\langle \text{patt} \rangle ::= \emptyset \mid \langle \text{ne-patt} \rangle$$

$$\langle \text{ne-patt} \rangle ::= \langle \text{strct} \rangle \mid \langle \text{strct} \rangle \, , \langle \text{ne-patt} \rangle$$

$$\langle \text{strct} \rangle ::= \langle \text{prim} \rangle \mid \left\{ \begin{matrix} \langle \text{ne-patt} \rangle \\ \langle \text{ne-patt} \rangle \\ \vdots \\ \langle \text{ne-patt} \rangle \end{matrix} \right\} \mid \left[\begin{matrix} \langle \text{ne-patt} \rangle \\ \langle \text{ne-patt} \rangle \\ \vdots \\ \langle \text{ne-patt} \rangle \end{matrix} \right] \mid \neg[\langle \text{ne-patt} \rangle] \mid$$

$$\neg\langle \text{strct} \rangle \mid (\langle \text{ne-patt} \rangle) a\text{-}b \mid (\langle \text{ne-patt} \rangle) a \mid (\langle \text{ne-patt} \rangle)$$

$$\langle \text{prim} \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \ldots \mid \mathbf{z}$$

Table 3.IX: The universe of patterns.

Universe:

$$\emptyset^U = \{\epsilon\}$$

$$x^U = \sigma \qquad \text{where } x \text{ is } \langle \text{prim} \rangle$$

$$(A\,,B)^U = A^U \sim B^U \qquad \text{where } A \text{ is } \langle \text{strct} \rangle \text{ and } B \text{ is } \langle \text{ne-patt} \rangle$$

$$(\left\{ \begin{matrix} A \\ B \\ \vdots \\ C \end{matrix} \right\})^U = A^U \cup B^U \cup \ldots \cup C^U$$

$$(\left[\begin{matrix} A \\ B \\ \vdots \\ C \end{matrix} \right])^U = A^U \cup B^U \cup \ldots \cup C^U$$

$$(\neg A)^U = A^U$$

$$\big((A)a\text{-}b\big)^U = \big(A^a\big)^U \cup \big(A^{a+1}\big)^U \cup \ldots \cup \big(A^b\big)^U$$

$$\big((A)a\big)^U = \big(A^a\big)^U \cup \big(A^{a+1}\big)^U \cup \ldots$$

$$\big((A)\big)^U = \{\epsilon\} \cup A^U$$

If the succeeding structure (**f**) were not concatenated to the individual arguments before the meaning is determined—as would be the case in a strict compositional definition—the string 'bff...' would match this pattern. This is not the kind of thing one wants to happen when 'bff' is excluded from pattern (3.26):

$$\neg\begin{Bmatrix}\mathbf{b}\\ \mathbf{c},\mathbf{d}\end{Bmatrix},\mathbf{f} \tag{3.26}$$

With the given definition (3.24), however, it can be seen fairly easily that (3.25) will exclude at least the same strings as (3.26):

$$\underline{\begin{bmatrix}\mathbf{(cons)1\text{-}2}\\ \neg\begin{Bmatrix}\mathbf{b}\\ \mathbf{c},\mathbf{d}\end{Bmatrix}\end{bmatrix},\mathbf{f},\otimes} = \underline{\mathbf{(cons)1\text{-}2},\mathbf{f},\otimes}\cap\underline{\neg\begin{Bmatrix}\mathbf{b}\\ \mathbf{c},\mathbf{d}\end{Bmatrix},\mathbf{f},\otimes} \tag{3.27}$$

The second term of the righthand side of (3.27) equals pattern (3.26).

3.7.2 Relation to the compositional formalism

The given definition is thus consistent with the philosophy of excluding explicit nofits. There is, however, a complication with simultaneity which does not occur with complementation and alternation. Consider, for instance, pattern (3.28):

$$\begin{bmatrix}\mathbf{a}\\ \mathbf{a},\mathbf{t}\end{bmatrix},\mathbf{t} \tag{3.28}$$

According to the definition, (3.24), this pattern denotes all strings starting with 'att':

$$\left\langle\begin{bmatrix}\mathbf{a}\\ \mathbf{a},\mathbf{t}\end{bmatrix},\mathbf{t}\right\rangle = \underline{\mathbf{a},\mathbf{t},\otimes}\cap\underline{\mathbf{a},\mathbf{t},\mathbf{t},\otimes} = \underline{\mathbf{a},\mathbf{t},\mathbf{t},\otimes} = \langle\mathbf{a},\mathbf{t},\mathbf{t}\rangle \tag{3.29}$$

This is an example of the property that simultaneity distributes over concatenation. This is not so surprising since it is defined that way:

$$\underline{\begin{bmatrix}A\\ B\\ \vdots\\ C\end{bmatrix},Y} = \underline{A,Y}\cap\underline{B,Y}\cap\ldots\cap\underline{C,Y} = \underline{\begin{bmatrix}A,Y\\ B,Y\\ \vdots\\ C,Y\end{bmatrix}}$$

However, the point is that, as opposed to alternation, simultaneity does not distribute over concatenation in the compositional case. In the compositional case concatenation and simultaneity are defined as follows[4]:

$$\underline{\underline{A\,,B}} = \underline{\underline{A}} \sim \underline{\underline{B}}$$

$$\underline{\underline{\begin{bmatrix} A \\ B \\ \vdots \\ C \end{bmatrix}}} = \underline{\underline{A}} \cap \underline{\underline{B}} \cap \ldots \cap \underline{\underline{C}}$$

Thus:

$$\underline{\underline{\begin{bmatrix} \mathtt{a}, \left\{ \begin{matrix} \mathtt{t,e} \\ \mathtt{e} \end{matrix} \right\} \\ \mathtt{a,t}, \left\{ \begin{matrix} \mathtt{t,e} \\ \mathtt{e} \end{matrix} \right\} \end{bmatrix}}} = \underline{\underline{\mathtt{a}, \left\{ \begin{matrix} \mathtt{t,e} \\ \mathtt{e} \end{matrix} \right\}}} \cap \underline{\underline{\mathtt{a,t}, \left\{ \begin{matrix} \mathtt{t,e} \\ \mathtt{e} \end{matrix} \right\}}} = \underline{\underline{\mathtt{a,t,e}}}$$

while:

$$\underline{\underline{\begin{bmatrix} \mathtt{a} \\ \mathtt{a,t} \end{bmatrix}, \left\{ \begin{matrix} \mathtt{t,e} \\ \mathtt{e} \end{matrix} \right\}}} = \underline{\underline{\begin{bmatrix} \mathtt{a} \\ \mathtt{a,t} \end{bmatrix}}} \sim \underline{\underline{\left\{ \begin{matrix} \mathtt{t,e} \\ \mathtt{e} \end{matrix} \right\}}} = \emptyset \tag{3.30}$$

Therefore, beside the inconsistent patterns, this is a second case for which the proposed semantics differs from a compositional one. In this case, however, it is not so clear that (3.29) is a more expected or desirable interpretation than (3.30). On the other hand, it may be argued that patterns like these, which can be characterized by the fact that simultaneity operates on paths of different length, have very little intuitive meaning to begin with.

3.7.3 de Morgan's laws

Since the formalism now features an 'or', an 'and' and a 'not', one may wonder whether an equivalent of de Morgan's laws is valid. From the field of logics we know that $\neg(a \vee b) = \neg a \wedge \neg b$ and $\neg(a \wedge b) = \neg a \vee \neg b$. Is something equivalent valid in the semi-compositional formalism, in other words, are equations (3.31) and (3.32) valid?

[4]The double underlining serves to distinguish this definition from the proposed formalism.

$$\neg \left\{ \begin{matrix} A \\ B \end{matrix} \right\} = \left[\begin{matrix} \neg A \\ \neg B \end{matrix} \right] \tag{3.31}$$

$$\neg \left[\begin{matrix} A \\ B \end{matrix} \right] = \left\{ \begin{matrix} \neg A \\ \neg B \end{matrix} \right\} \tag{3.32}$$

It turns out that these equations are not valid in general. This can be seen, for instance, as follows:

$$\left\langle \neg \left\{ \begin{matrix} \mathbf{a} \\ \mathbf{o},\mathbf{u} \end{matrix} \right\}, \mathbf{t} \right\rangle = \underline{\neg \left\{ \begin{matrix} \mathbf{a} \\ \mathbf{o},\mathbf{u} \end{matrix} \right\}, \mathbf{t}, \otimes} =$$

$$\left\{ \begin{matrix} \mathbf{a} \\ \mathbf{o},\mathbf{u} \end{matrix} \right\}^{U} \underline{\sim \mathbf{t}, \otimes} \setminus \underline{\left\{ \begin{matrix} \mathbf{a} \\ \mathbf{o},\mathbf{u} \end{matrix} \right\} \sim \mathbf{t}, \otimes} = \sigma^{1,2} t \sigma^* \setminus (\mathrm{at}\sigma^* \cup \mathrm{out}\sigma^*) \Rightarrow$$

$$\mathrm{art}\sigma^* \in \left\langle \neg \left\{ \begin{matrix} \mathbf{a} \\ \mathbf{o},\mathbf{u} \end{matrix} \right\}, \mathbf{t} \right\rangle \tag{3.33}$$

$$\left\langle \left[\begin{matrix} \neg \mathbf{a} \\ \neg[\mathbf{o},\mathbf{u}] \end{matrix} \right], \mathbf{t} \right\rangle = \underline{\left[\begin{matrix} \neg \mathbf{a} \\ \neg[\mathbf{o},\mathbf{u}] \end{matrix} \right], \mathbf{t}, \otimes} =$$

$$\underline{\neg \mathbf{a}, \mathbf{t}, \otimes} \cap \underline{\neg[\mathbf{o},\mathbf{u}], \mathbf{t}, \otimes} = (\sigma t \sigma^* \setminus \mathrm{at}\sigma^*) \cap (\sigma\sigma t \sigma^* \setminus \mathrm{out}\sigma^*) \Rightarrow$$

$$\mathrm{art}\sigma^* \notin \left\langle \left[\begin{matrix} \neg \mathbf{a} \\ \neg[\mathbf{o},\mathbf{u}] \end{matrix} \right], \mathbf{t} \right\rangle \tag{3.34}$$

Here, $\sigma^{1,2}$ denotes all strings of length 1 or 2.

Thus, (3.33) and (3.34) disprove (3.31). The other law, (3.32) can be disproved in a similar way.

The equivalent of de Morgan's laws is thus not valid in general. It should be noted, however, that in the compositional formalism the two equations are not valid either[5]. This can be verified by taking $A = \mathbf{a}$ and $B = \mathbf{o},\mathbf{u}$.

[5]Only in a compositional formalism where the universe is defined as $U = \sigma^*$ will de Morgan's laws be valid:

$$\underline{\neg \left\{ \begin{matrix} A \\ B \end{matrix} \right\}} = \sigma^* \setminus (\underline{A} \cup \underline{B}) = (\sigma^* \setminus \underline{A}) \cap (\sigma^* \setminus \underline{B}) = \underline{\left[\begin{matrix} \neg A \\ \neg B \end{matrix} \right]}$$

The other law is proved analogously.

On the other hand, it is not so that the equations are always invalid. If, for instance, the lengths of all paths in the complemented structure are equal, it can be shown that the equations *are* valid. Thus, for instance:

$$\left\langle \neg \left\{ \begin{matrix} \mathbf{a} \\ \mathbf{o} \end{matrix} \right\}, \mathbf{t} \right\rangle = \left\langle \left[\begin{matrix} \neg\mathbf{a} \\ \neg\mathbf{o} \end{matrix} \right], \mathbf{t} \right\rangle \tag{3.35}$$

3.7.4 Complementing simultaneity

A final pattern which deserves some discussion is the use of complementation in co-ordination with simultaneity. Consider, for instance, pattern (3.36):

$$\begin{aligned} \left\langle \neg \left[\begin{matrix} \mathbf{a} \\ \mathbf{a},\mathbf{t} \end{matrix} \right], \mathbf{t} \right\rangle \;=\; & \left[\begin{matrix} \mathbf{a} \\ \mathbf{a},\mathbf{t} \end{matrix} \right]^{U} \sim\underline{\mathbf{t},\otimes} \setminus \underline{\left[\begin{matrix} \mathbf{a} \\ \mathbf{a},\mathbf{t} \end{matrix} \right] \sim\underline{\mathbf{t},\otimes}} = \\ & \left[\begin{matrix} \mathbf{a} \\ \mathbf{a},\mathbf{t} \end{matrix} \right]^{U} \sim\underline{\mathbf{t},\otimes} \setminus \emptyset = \sigma^{1,2}\mathrm{t}\sigma^{*} \end{aligned} \tag{3.36}$$

Here, the result is somewhat unexpected. One could expect the strings 'att...' to be excluded from this pattern, since they match pattern (3.29), which only differs from (3.36) in the absence of a complementation sign.

The reason for this phenomenon is that explicit nofits are defined as $\underline{A \sim B}$ rather than $\underline{A,B}$. Then, when $A = \emptyset$ the explicit nofits are empty, too. This does not necessarily have to be the case for $\underline{A,B}$, as exemplified in (3.29). This phenomenon of the explicit nofits being empty did not occur previously in simple complemented structures (no nested complementation) since there is no pattern for which $A = \emptyset$. So (3.36) is a simple complemented structure of which one may feel that the semi-compositional does not behave in the expected manner.

3.8 Discussion

In general, the phenomenon that not all strings are excluded which one would expect to be, occurs when $\underline{A \sim B} \neq \underline{A,B}$. Apart from simultaneous structures with paths of unequal lengths, this can only occur when A contains a complemented structure (also) containing paths of different lengths. So only when complementation is nested to a level of two or more does this phenomenon occur. This is exactly the reason why double complementation may not always be annihilated.

In this light it would be more consistent to define explicit nofits as $\underline{A}, B$. In fact, this is the most logical choice since $\underline{A}, B$ denotes precisely those structures which match $\neg A, B$ if the '$\neg$' sign is omitted. This would resolve the above particular case (3.36), but not all of the problems attached to this matter. Appendix 3.D discusses this option with regard to double complementation. The following is probably an even stronger argument against this option.

One of the main consequences of this choice is that complex patterns must be evaluated as a whole. When for instance a pattern is specified with complementation nested to a certain level, the full pattern is important to determine the meaning of the most deeply nested complementation, something which also holds for all the other complementations. This tends to get so complicated that the only way to evaluate patterns in which complementation is nested to level two or more is to write a computer program which patiently executes the definition. This is not conducive to quick design of complex patterns.

This is, in fact, a plea for compositionality. In a compositional system, the meaning of any pattern, however complex, is determined by the composing parts. The meaning of a part is not changed by altering something outside that part. The meaning of a pattern is determined from small to large rather than having to view the pattern as a whole to start with.

The current definition, with the explicit nofits defined as $\underline{A} \sim \underline{B}$, can be seen as a compromise between the two extremes. It resolves the objection to the compositional system which does not exclude explicit nofits. On the other hand, within complementation the succeeding structure is 'hidden', i.e., in determining the meaning of a complemented pattern one can separately determine A^U, $\underline{A}$ and $\underline{B}$, and combine these afterwards. When complementation is nested, the succeeding structure does not extend outside the complementation which is nested one level less deep. As we have seen, double complementation may for this reason not always be annihilated. On the other hand, all patterns can, with some practice, now be evaluated by hand, since the patterns can be decomposed into smaller parts which can be evaluated separately.

On theoretical grounds the semi-compositional formalism is thus not completely satisfactory. For practical purposes, however, it might be satisfactory, because the cases which are not resolved satisfactorily, such as double complementation, deeply nested complemented structures or complementing simultaneity which contains paths of different lengths, are highly unlikely to be used in practice. The semi-compositional definition excludes explicit nofits in the practical cases and has a semi-compositional behaviour for complex patterns. It thus seems a practical compromise between the conflicting requirements

of strict compositionality on the one hand and interpretation according to expectation on the other.

Based on these practical grounds it was decided to implement the semi-compositional formalism in TooLiP to investigate its merits. In the next chapter the implementation of the formalism will be discussed, and in the concluding chapter (among other things) the merits of the semi-compositional formalism will be considered.

3.9 Conclusion

In this chapter a formalism is proposed in which a complementation operator (the 'not') and a simultaneous operator (the 'and') are included in the formalism of regular expressions.

Regular expressions are defined in a compositional manner. This is an important theoretical and practical property, since the meaning of any expression, however complex, can be determined in parts, from small to large.

The inclusion of the new operators in regular expressions in a compositional manner has the practical objection that for a certain class of patterns, the so-called inconsistent patterns, the formal interpretation does not correspond to the impression the patterns create.

The main motivation to try and resolve this objection is of an ergonomical nature. The main users of the system in which the formalism must be implemented cannot in general be expected to be familiar with the details of the behaviour of such extended regular expressions.

The attempt to resolve this objection has resulted in a slightly different formalism, which can best be characterized as a semi-compositional formalism. Although the pattern succeeding a specific structure must be included in determining the meaning of that structure, it remains possible to decompose complex patterns into smaller parts, determine the meaning of each of those parts, and determine the meaning of the whole pattern from the meaning of the individual parts. The extent, however, to which the pattern may be decomposed is smaller than in the compositional case. In the semi-compositional case patterns can be decomposed only with respect to complementation, whereas in the compositional case patterns can be decomposed with respect to all structures.

In defining the semi-compositional formalism three essential choices have been made. The first one concerns the choice of universe. Rather than taking the set of all strings, σ^*, as universe, a character counting mechanism has been defined. As a consequence, the user does not explicitly have to specify

the desired universe for each complemented structure that is used. On the other hand, the validity of the equivalent for de Morgan's laws is lost. Both the compositional and the semi-compositional formalism described in this chapter feature the character counting universe.

The second choice is to exclude explicit nofits. This is the main difference between the semi-compositional and the compositional formalism. As a consequence strict compositionality is lost, but the patterns behave more to expectation.

The third choice is the definition of explicit nofits. In the semi-compositional formalism which is proposed here this is defined as $\underline{A}\sim\underline{B}$, whereas the more logical choice would be $\underline{A\,,B}$. As a consequence the semi-compositional formalism still does not always behave as expected, i.e., not always are the 'real' explicit nofits $(\underline{A\,,B})$ excluded. On the other hand, compositionality is not all together lost.

From a theoretical point of view the semi-compositional formalism is thus not completely satisfactory. For practical purposes, however, it might be satisfactory, because the cases for which it does not behave satisfactorily are highly unlikely to be used in practice. The semi-compositional definition excludes explicit nofits in the practical cases and has a semi-compositional behaviour for complex patterns. It thus seems a practical compromise between the conflicting requirements of practical needs and theoretical elegance. Based on these practical grounds it has been decided to implement the semi-compositional formalism in TooljP.

Appendix 3.A
Distributivity of patterns

In this appendix it is proved that, given the semantics of Table 3.II or 3.V and the definition of string concatenation, alternation is distributive over concatenation for patterns.

Propositions (A–1) and (A–2) hold in general for sets of strings.

$$(A \cup B)\sim C = (A\sim C) \cup (B\sim C) \tag{A–1}$$

$$A\sim(B \cup C) = (A\sim B) \cup (A\sim C) \tag{A–2}$$

Proof:

$$\begin{array}{l}(A \cup B)\sim C \stackrel{\text{def}}{=} \{d_1 d_2 \mid d_1 \in (A \cup B) \wedge d_2 \in C\} = \\ \{d_1 d_2 \mid (d_1 \in A \wedge d_2 \in C) \vee (d_1 \in B \wedge d_2 \in C)\} = \\ \{d_1 d_2 \mid d_1 \in A \wedge d_2 \in C\} \cup \{d_1 d_2 \mid d_1 \in B \wedge d_2 \in C\} = \\ (A\sim C) \cup (B\sim C)\end{array}$$

(A–2) is proved analogously.

From these general properies of string sets the distributivity of concatenation over alternation for patterns follows directly:

$$\begin{array}{rcl}\underline{\left\{\begin{array}{c}X\\Y\\\vdots\\Z\end{array}\right\}, B} & = & \underline{(X \cup Y \cup \ldots \cup Z)\sim B} \\ & \stackrel{\text{(A–1)}}{=} & \underline{X\sim B \cup Y\sim B \cup \ldots \cup Z\sim B} \\ & = & \underline{X, B \cup Y, B \cup \ldots \cup Z, B} \\ & = & \underline{\left\{\begin{array}{c}X, B\\Y, B\\\vdots\\Z, B\end{array}\right\}}\end{array} \tag{A–3}$$

Similarly,

$$\underline{A, \left\{\begin{array}{c}X\\Y\\\vdots\\Z\end{array}\right\}} \stackrel{\text{(A–2)}}{=} \underline{\left\{\begin{array}{c}A, X\\A, Y\\\vdots\\A, Z\end{array}\right\}} \tag{A–4}$$

Appendix 3.B
Simplification of complementation

In this appendix it is proved that the definition of complementation as given in (3.15) can be simplified to (3.16):

$$(A^U \setminus \underline{A})\sim\underline{B} \setminus \underline{A}\sim\underline{B} = A^U\sim\underline{B} \setminus \underline{A}\sim\underline{B} \tag{B-1}$$

Proof:

$$\begin{aligned} X &= \big((P \setminus A)\sim B\big) \setminus (A\sim B) \\ Y &= (P\sim B) \setminus (A\sim B) \end{aligned}$$

$$\begin{aligned} d \in X &\Rightarrow \begin{cases} \exists d_1, d_2 \mid d = d_1 d_2, d_1 \in P, d_1 \notin A, d_2 \in B \\ d \notin A\sim B \end{cases} \\ &\Rightarrow \begin{cases} \exists d_1, d_2 \mid d = d_1 d_2, d_1 \in P, d_2 \in B \\ d \notin A\sim B \end{cases} \\ &\equiv d \in Y \end{aligned}$$

$$\begin{aligned} d \in Y &\Rightarrow \begin{cases} \exists d_1, d_2 \mid d = d_1 d_2, d_1 \in P, d_2 \in B \\ d \notin A\sim B \end{cases} \\ &\Rightarrow \begin{cases} \exists d_1, d_2 \mid d = d_1 d_2, d_1 \in P, d_2 \in B \\ \forall d_1, d_2 \neg(d = d_1 d_2 \wedge d_1 \in A \wedge d_2 \in B) \\ d \notin A\sim B \end{cases} \\ &\Rightarrow \begin{cases} \exists d_1, d_2 \mid d = d_1 d_2, d_1 \in P, d_1 \notin A, d_2 \in B \\ d \notin A\sim B \end{cases} \\ &\equiv d \in X \end{aligned}$$

Appendix 3.C
Equivalence of semantics

In this appendix it is proved that for consistent patterns the semantics given by Table 3.V and Table 3.VI are equivalent. For this purpose, the semantics of Table 3.V is referred to by a single underlining, the semantics of Table 3.VI by a double underlining.

First, the formal definition of consistent and inconsistent patterns is given. Next, given the consistency of a pattern, the equivalence of the two semantics are shown. Finally, two characteristics of appearance, which guarantee consistency, are discussed.

Definition: A pattern X is consistent if $Cons(X) = \mathbf{true}$, and inconsistent if $Cons(X) = \mathbf{false}$.

$Cons(X)$ is given by:

$$Cons(\emptyset) = \mathbf{true}$$

$$Cons(\otimes) = \mathbf{true}$$

$$Cons(X) = Cons(X, \emptyset)$$

$$\begin{array}{lll}
Cons(X, Y) = & \text{if } X = \emptyset & \text{then } Cons(Y) \\
& \text{if } X = \langle \text{prim} \rangle & \text{then } Cons(Y) \\
& \text{if } X = \left\{ \begin{array}{c} A \\ B \\ \vdots \\ C \end{array} \right\} & \text{then } Cons(A, Y) \wedge Cons(B, Y) \\
& & \quad \wedge \ldots \wedge Cons(C, Y) \\
& \text{if } X = \neg A & \text{then } Cons(A) \wedge Cons(Y) \\
& & \quad \wedge \left[(A^U \setminus \underline{\underline{A}}) \sim \underline{\underline{Y}} \cap \underline{\underline{A}} \sim \underline{\underline{Y}} = \emptyset \right]
\end{array}$$

For consistent patterns the semantics of Table 3.V and the semantics of Table 3.VI yield the same results. This can be shown as follows:

$$\underline{\underline{\emptyset}} = \{\epsilon\} = \underline{\underline{\emptyset}}$$

$$\underline{\underline{\otimes}} = \sigma^* = \underline{\underline{\otimes}}$$

$$\underline{\underline{x}} = \{x\} = \{x\}\sim\{\epsilon\} = \{x\}\sim\underline{\underline{\emptyset}} = \underline{\underline{x,\emptyset}} = \underline{\underline{x}}$$

$$\underline{\left\{\begin{matrix} A \\ B \\ \vdots \\ C \end{matrix}\right\}} = \underline{A} \cup \underline{B} \cup \ldots \cup \underline{C} = \underline{A,\emptyset} \cup \underline{B,\emptyset} \cup \ldots \cup \underline{C,\emptyset} \overset{\text{ind.}}{=}$$

$$= \underline{\underline{A,\emptyset}} \cup \underline{\underline{B,\emptyset}} \cup \ldots \cup \underline{\underline{C,\emptyset}} = \underline{\underline{\left\{\begin{matrix} A \\ B \\ \vdots \\ C \end{matrix}\right\}}}$$

$$\underline{\neg A} = A^U \setminus \underline{A} \overset{\text{ind.}}{=} A^U \setminus \underline{\underline{A}} = A^U\sim\underline{\underline{\emptyset}} \setminus \underline{\underline{A}}\sim\underline{\underline{\emptyset}} = \underline{\underline{\neg A,\emptyset}} = \underline{\underline{\neg A}}$$

$$\underline{A,B} = \text{if } A = \emptyset \rightarrow \underline{B} \overset{\text{ind.}}{=} \underline{\underline{B}} = \underline{\underline{A,B}}$$

$$\text{if } A = x \rightarrow \{x\}\sim\underline{B} \overset{\text{ind.}}{=} \{x\}\sim\underline{\underline{B}} = \underline{\underline{A,B}}$$

$$\text{if } A = \underline{\left\{\begin{matrix} X \\ Y \\ \vdots \\ Z \end{matrix}\right\}} \rightarrow (\underline{X} \cup \underline{Y} \cup \ldots \cup \underline{Z})\sim\underline{B}$$

$$\overset{\text{(A-1)}}{=} \underline{X}\sim\underline{B} \cup \underline{Y}\sim\underline{B} \cup \ldots \cup \underline{Z}\sim\underline{B}$$

$$= \underline{X,B} \cup \underline{Y,B} \cup \ldots \cup \underline{Z,B}$$

$$\overset{\text{ind.}}{=} \underline{\underline{X,B}} \cup \underline{\underline{Y,B}} \cup \ldots \cup \underline{\underline{Z,B}}$$

$$= \underline{\underline{A,B}}$$

$$\text{if } A = \neg X \rightarrow (X^U \setminus \underline{X})\sim\underline{B} \overset{\text{ind.}}{=} (X^U \setminus \underline{\underline{X}})\sim\underline{\underline{B}}$$

$$\overset{\text{cons.}}{=} (X^U \setminus \underline{\underline{X}})\sim\underline{\underline{B}} \setminus \underline{\underline{X}}\sim\underline{\underline{B}}$$

$$\overset{\text{(B-1)}}{=} X^U\sim\underline{\underline{B}} \setminus \underline{\underline{X}}\sim\underline{\underline{B}} = \underline{\underline{\neg X,B}}$$

$$= \underline{\underline{A,B}}$$

Characteristics of appearance

If a pattern consists exclusively of positive structures, it follows directly from the definition that it is consistent.

If a pattern contains a complemented structure which contains exclusively patterns of a certain specific length, it can be shown that this pattern is consistent:

1) d is a candidate for $\neg X, Y \Rightarrow \exists\, d_1 d_2 \mid d_1 \in X^U \setminus \underline{X} \wedge d_2 \in \underline{Y}$
$d_1' d_2' = d_1 d_2 = d \quad \rightarrow$

   a) $|d_1'| \neq |d_1| \overset{\text{spec.length}}{\Rightarrow} d_1' \notin \underline{X} \Rightarrow$
   no reason for $d_1' d_2' \in \underline{X} \sim \underline{Y}$

   b) $|d_1'| = |d_1| \Rightarrow d_1' = d_1 \Rightarrow d_1' \notin \underline{X} \Rightarrow$
   no reason for $d_1' d_2' \in \underline{X} \sim \underline{Y}$

   $\Rightarrow \quad d_1' d_2' = d_1 d_2 = d \notin \underline{X} \sim \underline{Y}$

2) d is an explicit nofit for $\neg X, Y \Rightarrow \exists\, d_1 d_2 \mid d_1 \in \underline{X} \wedge d_2 \in \underline{Y} \Rightarrow$
$d_1 \notin X^U \setminus \underline{X} \Rightarrow d_1 d_2 \notin X^U \setminus \underline{X} \sim \underline{Y}$

$\Rightarrow \quad \underline{X} \sim \underline{Y} \cap X^U \setminus \underline{X} \sim \underline{Y} = \emptyset \Rightarrow \neg X, Y =$ consistent.

Appendix 3.D
Alternative formalisms

In this appendix some alternatives for defining complementation are discussed. A drawback of the semantics of Table 3.VI is that double negation may not unconditionally be annihilated. The reason for this is that complementation on the inner level loses track of the succeeding structure of the outer level. Therefore, the inner complementation cannot fully compensate for the explicit nofits of the outer complementation. For instance, consider pattern (D–1):

$$\neg\neg\left\{\begin{matrix}\mathbf{a}\\ \mathbf{o,u}\end{matrix}\right\},\mathbf{t} \tag{D–1}$$

The inner complementation, $\neg\{\begin{smallmatrix}\mathbf{a}\\ \mathbf{o,u}\end{smallmatrix}\}$, does not see the succeeding structure **t**. Therefore, the string 'at' will not be excluded from $\neg\{\begin{smallmatrix}\mathbf{a}\\ \mathbf{o,u}\end{smallmatrix}\}$. Then, 'at~t...' is an explicit nofit of the outer complementation, and thus not an element of the set of strings denoted by (D–1).

A solution to this could be to redefine complementation so that the inner complementation can also see the succeeding structure of the outer complementation. This can be done by defining complementation as follows:

$$\underline{\neg A,B} = A^{U}\sim\underline{B}\setminus\underline{A,B} \tag{D–2}$$

Compared with the proposed definition (3.16), the two only differ in the set which is subtracted from the generating set $A^{U}\sim\underline{B}$. It is as if the explicit nofits have been redefined. With this definition of complementation one can prove that a double complementation may always be annihilated:

$$\begin{aligned}\underline{\neg\neg A,B} &= (\neg A)^{U}\sim\underline{B}\setminus\underline{\neg A,B} = A^{U}\sim\underline{B}\setminus(A^{U}\sim\underline{B}\setminus\underline{A,B})\\ &= \underline{A,B}\cap A^{U}\sim\underline{B} = \underline{A,B}\end{aligned} \tag{D–3}$$

However, this interpretation of complementation can lead to unexpected results. Consider the pattern $\neg A,B$, where

$$A=\left\{\begin{matrix}X\\ Y\end{matrix}\right\}\quad X=\neg\left\{\begin{matrix}\mathbf{a}\\ \mathbf{o,u}\end{matrix}\right\}\quad Y=\neg\left\{\begin{matrix}\mathbf{e}\\ \mathbf{a,t}\end{matrix}\right\}\quad\text{and}\quad B=\mathbf{t},\otimes \tag{D–4}$$

This leads to pattern (D–5):

$$\neg\left\{\begin{array}{l}\neg\left\{\begin{array}{c}\mathbf{a}\\ \mathbf{o},\mathbf{u}\end{array}\right\}\\ \neg\left\{\begin{array}{c}\mathbf{e}\\ \mathbf{a},\mathbf{t}\end{array}\right\}\end{array}\right\},\mathbf{t},\otimes \qquad \text{(D–5)}$$

Applying (D–2) to interpret this pattern, (D–5) will include for instance the strings 'att...'. However, if we examine the candidates of $\neg A, B$, we find that there are none; the universe of the structure (A^U) is equal to the meaning of the structure $(\underline{A})$, the difference $(A^U \setminus \underline{A})$ being empty. It seems awkward that the new complementation includes strings which are not candidates in the first place.

The reason for this is that the simplification of (3.15) into (3.16) is not valid if explicit nofits are defined differently. In other words, the generating set of (3.16) as compared with the generating set of (3.15) includes strings which are not excluded by $\underline{A, B}$.

A logical way to overcome this objection is to redefine complementation as follows:

$$\underline{\neg A, B} = (A^U \setminus \underline{A}) \sim \underline{B} \setminus \underline{A, B} \qquad \text{(D–6)}$$

Here, we start by selecting the candidates, rather than the strings which satisfy the length condition as required by the universe, so the objection to the previous attempt is automatically removed.

Unfortunately, (D–6) does not satisfy the double complementation requirement. This can be seen as follows:

$$\begin{aligned}\underline{\neg\neg A, B} &\overset{\text{(D–6)}}{=} \left[((\neg A)^U \setminus \underline{\neg A}) \sim \underline{B}\right] \setminus \underline{\neg A, B}\\ &= \left[A^U \setminus (A^U \setminus \underline{A}) \sim \underline{B}\right] \setminus \left[(A^U \setminus \underline{A}) \sim \underline{B} \setminus \underline{A, B}\right]\\ &= \underline{A} \sim \underline{B} \setminus \left[(A^U \setminus \underline{A}) \sim \underline{B} \setminus \underline{A, B}\right] \qquad \text{(D–7)}\end{aligned}$$

Let $A = \left\{ \begin{array}{l} \neg\{\mathbf{a} \\ \quad \mathbf{o},\mathbf{u}\} \\ \neg\{\mathbf{e} \\ \quad \mathbf{a},\mathbf{t}\} \end{array} \right\}$ and $B = \mathbf{t}, \otimes$.

Then:

$$\left.\begin{array}{l} \mathrm{att}\sigma^* \in \underline{A}\sim\underline{B} \\ \mathrm{att}\sigma^* \notin \underline{(A^U \setminus A)}\sim\underline{B} \\ \mathrm{att}\sigma^* \notin \underline{A,B} \end{array}\right\} \Rightarrow \left.\begin{array}{l} \mathrm{att}\sigma^* \in \underline{\neg\neg A, B} \\ \mathrm{att}\sigma^* \notin \underline{A,B} \end{array}\right\} \Rightarrow \underline{\neg\neg A, B} \neq \underline{A,B}$$

Thus, when complementation is defined as in (D–6), double complementation may not always be annihilated. It must be noted, however, that in the semi-compositional formalism, where explicit nofits are defined as $\underline{A}\sim\underline{B}$, this effect is more frequent than for the last definition. In the semi-compositional case this occurs for all inconsistent patterns, i.e., when $\underline{A,B} \neq \underline{A}\sim\underline{B}$. When definition (D–6) is applied this only occurs when $\underline{A^U} = \underline{A}$ or $\underline{A} = \emptyset$. These last two cases will probably never occur in practical situations.

As is argued in the main section of the chapter, (D–6) is the most consistent choice in the philosophy of excluding explicit nofits. A further discussion on this matter is included in section 3.8. For this appendix it suffices to conclude the two attempts illustrate the dilemma: either unexpected results are yielded for certain patterns, or the double complementation may not always be annihilated.

Other attempts also do not seem very promising, either. In the philosophy of excluding explicit nofits no other options seem available. Thus either one must try and find a solution in a totally different direction or accept that the property of being allowed to annihilate double complementation is lost. Since double complementation will probably not occur very often in practical situations (the simple positive statement is more transparent), and other attempts will probably violate the philosophy of excluding explicit nofits, it was decided to accept the loss of this property.

Chapter 4

Some aspects of the implementation of TooLjiP

Abstract

In this chapter those aspects of the implementation of TooLjiP are described which concern the process of matching patterns to the input, where input should be understood in the general sense of synchronized buffers.

For this purpose first the internal representation of patterns is discussed. The user-specified patterns are transformed into a dynamic data structure which is accessible for the matching routine. The dynamic structure codes the structure of the patterns, but some simple adjustments have been made also, which facilitate pattern matching during run-time.

Next, the algorithms which perform the pattern matching are presented. First the situation of a single input buffer is considered. In view of this input situation most of the functions for matching a particular structure in a pattern are given. Special attention is paid to the function for matching the complementation operator, since its definition gives rise to some additional computational complexity.

Then the more general situation of synchronized buffers is considered. The algorithm for matching primitives is somewhat altered in this situation. Since the synchronization mechanism is important for this routine, two possible synchronization mechanisms are discussed and compared. The more general one is chosen to be implemented and the buffer switching algorithm is given.

On the whole, with respect to the processing of patterns, TooLjiP can be viewed as a compiler/interpreter. The user-defined patterns are compiled from high-level source code to an internal representation. The internal representation is then interpreted by the functions for pattern matching.

4.1 Introduction

THE previous two chapters served to give a more or less complete functional specification of TooIJP. In chapter 2 the overall architecture is discussed, together with the mechanisms available to define an arbitrary conversion scheme. In chapter 3 the kernel of the linguistic rule, the pattern, is discussed, and its meaning is formalized in Table 3.VII. Together, the two chapters fully describe TooIJP's behaviour.

However, not only has TooIJP's behaviour been defined, TooIJP has also been implemented and has been operational in evolving versions since 1986. A chapter on the implementation may therefore not be omitted from its description. This chapter deals with those implementation aspects. However, no attempt has been made to cover all aspects of the implementation completely, since that would be a rather technical and tedious matter. Only the more important parts are dealt with.

To be precise: this chapter deals with those aspects of the implementation which have to do with matching a pattern to the input. Here, 'input' should be understood in a general sense, it is not necessarily the input given by the user. It can also be an intermediate result of a module, or the synchronized results of several modules. The function which matches patterns to the input is the kernel of the system: a linguistic rule is evaluated by examining the patterns of the focus, left and right contexts successively; the result of a module is determined by repetitive application of rules; the overall conversion, in turn, is determined by successive execution of the modules.

The implementation of this structured repetition will not be described. The exact nature of this structure has been described in detail in chapter 2 and its implementation is rather straightforward; each function (such as evaluating a rule) is embedded in the function that directly needs it (such as executing the module). The algorithm closely resembles the functional specification given in Appendix 2.A. Since it does not seem very useful to repeat this in detail once again, that part of the implementation is not included.

As to pattern matching, TooIJP contains functions which are similar to those found in compilers. The patterns, which are specified in a certain user-friendly notation, often called source code, are parsed and represented internally, before actual pattern matching takes place. However, some functions are not implemented according to standard compiler techniques, such as given by Aho, Sethi & Ullman (1986). The functionality of the system is, of course, of primary interest. Moreover, TooIJP features extensions to the standard theory, the extended regular expressions, in respect of which it is not obvious how automata can be constructed to evaluate them. Throughout the chapter,

however, the relation to the standard theory and techniques will be indicated where possible.

Thus, the main topic of this chapter is how patterns are matched against input (in the general sense). For this purpose it is first discussed how the patterns specified by the user in the linguistic source files are represented internally (section 4.2). Then, the strategy of matching these (internally represented) patterns to the input is formulated explicitly in algorithmic-like structures (section 4.3). The matching process is initially derived for the special case of a single input buffer (for instance an intermediate module result). Derivational history cannot yet be accessed. The general case of synchronized input buffers, which provide this information, is the topic of section 4.4. A general discussion on the characteristics of TooJiP—as far as the implementation is concerned—is included in section 4.5. Finally, the most important conclusions of this chapter are summarized in the last section.

4.2 The internal representation of patterns

In almost all computer applications there is a conversion phase of the instructions a user has specified in source code to a computer internal code (object code). For instance, programs written in Pascal should be compiled first before they can be executed. In TooJiP there also exists a high-level source code, the linguistic rules. These are coded in a format which adheres as closely as possible to the linguist's wishes. Just like source code in other applications, the linguistic rules are also converted to an internal format. This contains the same information but is more efficiently processed by the computer.

Fig. 4.1 depicts this process. The high-level linguistic rules are compiled into an internal representation, which is input to the pattern matcher. In this case the internal representation is a dynamic data structure rather than a sequence of machine instructions. The data structure reflects the organization of the linguistic input in a way which is accessible more quickly for a computer program.

In this section it is discussed what the internal representation of patterns looks like. For this purpose first an informal strategy for matching patterns is formulated. With the informal strategy in mind we will then consider the internal representation of patterns. The construction of the compiler which translates the source code into this representation is not discussed, as it closely resembles the parsing phase of a compiler (see for instance Aho *et al.* (1986)).

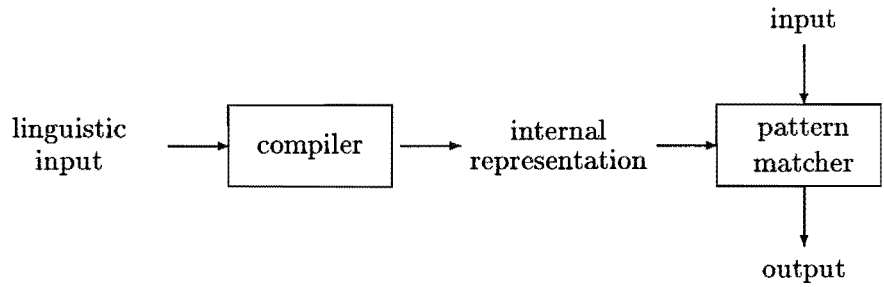


Figure 4.1: More detailed inside view of TooJiP. The linguistic input is compiled into an internal representation, which in turn is input to the pattern matcher.

4.2.1 An informal matching strategy

The semantics, which is derived in chapter 3 and given in Table 3.VII, provides a meaning to a pattern. To be precise, it defines the set of strings which are denoted by the pattern. It is the kind of thing a linguist is interested in, since when he specifies a pattern, he must be able to determine which strings match that pattern. What we shall be considering throughout this chapter is an algorithm which determines whether or not an arbitrary input string matches an arbitrary pattern, that is, whether or not the input is part of the set denoted by the pattern.

In general, the set denoted by a pattern is not finite; the pattern describes a set of strings which 'start' in a certain manner; in other words, the head of the string must satisfy certain requirements. It is therefore not a very promising approach to try and determine the full set of strings and then determine whether the input is part of that set. Instead, it seems more appropriate to start with the actual input string and try to establish whether or not it is one of the possible heads.

From the syntax, Table 3.VIII, it can be seen that a non-empty pattern is a concatenation of structures. In the semantics the concatenation discriminates between the different possibilities for the leftmost structure. Five possible structures can be encountered, if we exclude the trivial case of empty patterns. They are discussed in relation to the matching routine.

If a primitive is used, one of the segments it denotes should be found at the appropriate place in the input and it should be followed by a string denoted by the succeeding pattern (this is the definition of string concatenation). Therefore, if such a segment is not found, we can stop the matching procedure, since the current input string can never lead to the desired head

of the strings denoted by the pattern. Only when the segment matches the primitive do we have to continue matching.

If an alternative structure is specified, the individual arguments of the structure should be concatenated to the succeeding pattern before the meaning of the thus formed patterns is determined, as explained above. For the matching routine this means that we can test for these patterns one by one, terminating with a positive result if one of the patterns is found, since they are ordered in an 'or' relation.

The simultaneous structure is similar to the alternative structure, save for the fact that the current input string should be present in all concatenated sub-patterns. Therefore, the same strategy as above can be followed, only in this case the matching fails if the string does not match a sub-patterns.

The strategy which is to be applied inside the complemented structure differs from the strategy described above. So far, when a primitive does not match, we can conclude that the path of concatenated primitives which we are considering does not match. We can start matching the next path in the case of alternatives, or conclude that the whole pattern does not match in the case of simultaneity. Inside complementation, however, a mismatch of a primitive will lead to the conclusion that that part of the complementation matches, which may—but does not necessarily have to—lead to a positive result value of the pattern as a whole. On the other hand, if a primitive matches inside a complemented structure, this may but does not necessarily have to lead to a negative result value of the pattern. Therefore, in both cases further processing is necessary as no direct conclusion can be drawn, and so the matching may not terminate in either case.

The optional structure is actually a shorthand notation for an alternative structure. It saves coding time for the users and improves legibility of the rules. Since the optional structure is only a notational shorthand for a certain alternative structure, the optional structure (for normal, positive patterns) can internally be represented as an alternative structure. This includes patterns denoting infinite repetition. Inside complemented patterns, however, this is not appropriate. As explained above, the matching routine must fully consider all cases inside complementation, which in the case of infinity would lead to non-termination. Therefore, a different internal representation is needed in the case where optionality is used inside complementation.

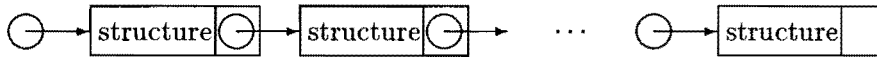
Thus, in positive structures, we may terminate the matching of the current path when a primitive does not match the input. Inside complemented structures this is not the case. Amongst other things a separate internal representation is then needed for optionality. With these characteristics in mind we can now take a closer look at the internal representation of patterns.

4.2.2 The representation

The internal representation of patterns is a modified syntax tree. A syntax tree is a hierarchical structure which accounts for the relationship of the elements in an expression (in this case a pattern). The modification involves some simple adjustments which can take place at compile-time, so as to speed up the run-time performance.

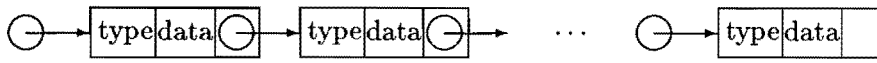
Two basic data structure concepts are used to represent patterns internally. These are the record type, in which one can join elements of arbitrary types into a compound type, and the linked list, in which one can store a list (an arbitrary number) of elements (for instance a record) and access them sequentially. As illustrated below, the record type is portrayed by a rectangular box, possibly sub-divided, while the linked list is represented by an open circle with an arrow (the link) pointing at an element, a rectangular box, which contains a link to the next element. The end of the list is represented by a filled circle.

As follows from the syntax, a pattern is a concatenation of structures. The number of concatenations is free, and varies significantly in practical situations. Therefore, a linked list of structures is an elegant way to represent patterns, rather than a fixed array. Each link represents a concatenation, each element of the list represents a structure:



Note that the filled circle, which indicates the end of the list, can be interpreted as the $\otimes$-pattern, to which all strings match. So as soon as this point is reached, one may conclude that the pattern which is searched for is present.

A structure can be any of the given five types. The kind of information a type represents differs per type. As only one type at a time can be used at a certain place, the different types of linguistic data (graphemes, phonemes, features, etc.) can be stored at the same place, provided an additional indication of which type is used. In Pascal terms this is called a variant record. The record which represents a structure then contains three fields: a type indication, the linguistic data, and a link field:

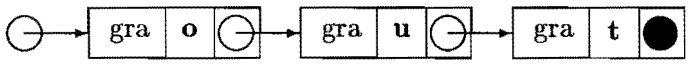


The internal representation will now be discussed for all possible structures. This comprises the four types of primitives: graphemes, phonemes,

grapheme features and phoneme features, and the four 'real' structures: alternation, simultaneity, optionality and complementation.

Graphemes

Graphemes refer to orthography, and can be any printable character. For instance, the pattern **o , u , t** (throughout the chapter all example patterns are printed in bold face) is internally represented as follows:



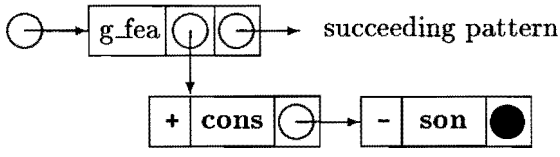
Phonemes

Phonemes refer to pronunciation. They are defined by the user and consist of a limited set. They are represented in a way similar to graphemes. For instance, the pattern **SJ , OO** is represented as follows:

Grapheme features

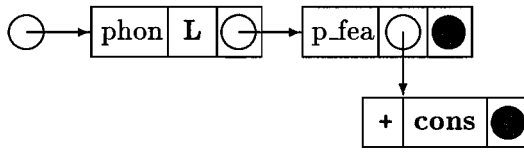
Grapheme features are user-defined, and are used to describe common properties of graphemes. A feature specification can consist of any positive (nonzero) number of features, each of which has a value ('+' or '−') and a name ('cons', 'son', etc.). For example, the feature specification **<+cons, -son>** denotes all consonantal graphemes which are not sonorant. Since the number of features used in a specification varies per rule, a linked list of feature elements is appropriate to represent them. The representation, for instance for the above specification, is as follows:

Used as part of a pattern, the linked list representing the feature specification forms the data part of the structure, as illustrated below:



Phoneme features

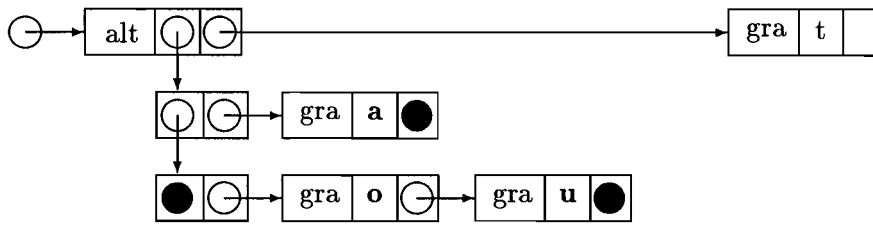
Phoneme features describe common properties of phonemes, analogous to grapheme features. These, too, follow from the definition tables. Their representation is equal to grapheme features, save for the type indication. So for instance, the pattern **L,<+CONS>**, which refers to a sequence of the phoneme /L/ followed by a consonantal phoneme, is represented as follows:



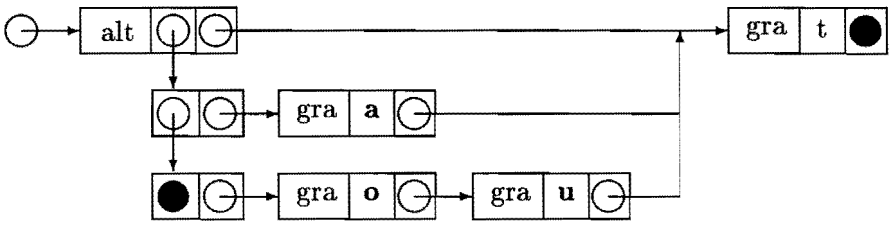
Alternative structures

An alternative structure can have any number of arguments, each of which can be a non-empty pattern. In accordance with the earlier solutions, this can be represented elegantly by a linked list. In this case, the elements of the linked list are patterns, i.e., linked lists of structures. Thus, here we see the same recursion in the internal representation as is present in the syntax.

The true syntax tree of the pattern $\{^{\mathbf{a}}_{\mathbf{o,u}}\}$**,t** (presented in the way patterns are represented) is as follows:



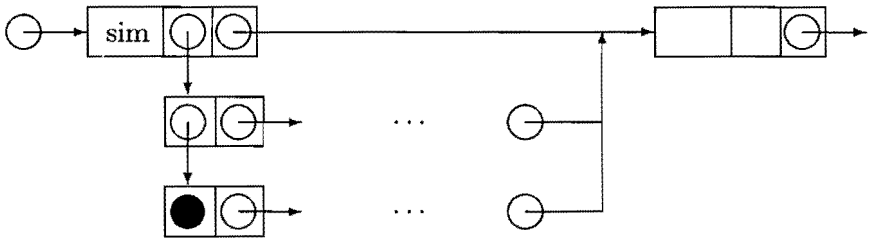
However, the semantics prescribe that the succeeding pattern should be concatenated to each alternative to form new patterns, whose meaning then should be determined. This transformation can be performed explicitly in the compiling phase by linking the end of each alternative to the succeeding structure. Thus, the above representation is adjusted to the following:



The original link, from the element marked as 'alt' to the succeeding structure is now spurious, but on the other hand it does no harm, so it may as well remain present. To achieve this representation, the compiler routine which constructs the internal representation must perform some additional computations, of course. Before parsing the arguments of the alternative structure, first the continuation entry for the succeeding structure should be computed, so it can be patched at the end of each argument. Per rule, this only needs to be done once. Moreover, constructing the internal representation is done off-line, as a preparation phase for the pattern matcher, so this does not affect the run-time performance.

Simultaneous structures

The simultaneous structure is similar to the alternative structure, be it that the interpretation differs. As this is the task of the matching routine, the internal representation for simultaneity only differs in the type indication from the alternative structure:

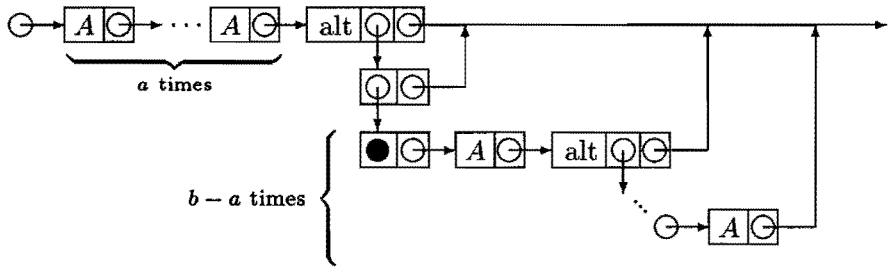


Optional structures

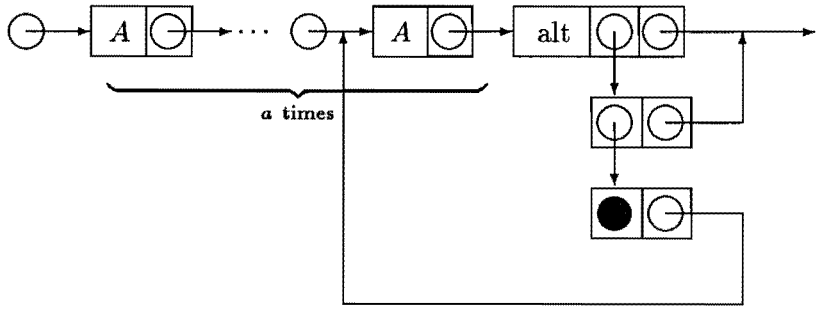
As previously explained in chapter 3, an optional structure is generally characterized by $(A)a\text{-}b$, where the parentheses denote the optional structure, A denotes an arbitrary pattern, and a and b denote the minimum and maximum number of times the pattern is required. Here, $a \leq b$, and a and b may be any non-negative number. One may, however, use a notational shorthand and omit either b or both a and b. In the first case, the pattern expresses infinite

repetition, that is, it should be present at least a times, but may be present any higher number of times. In the second case, the structure is interpreted in the true optional sense: it may be present or not, so it is as if $a = 0$ and $b = 1$.

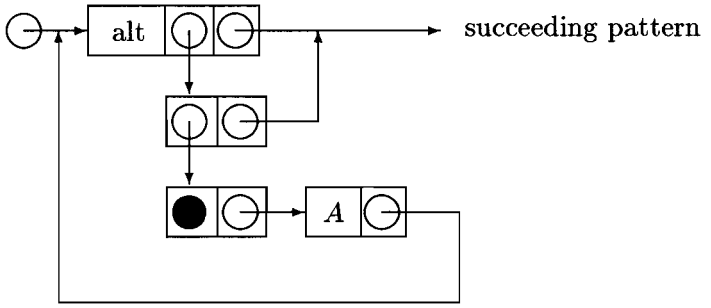
As mentioned, in normal positive structures it can be represented by an alternative structure. This saves the need to code a separate routine for positive optional structures. If b is limited, the pattern $(A)a\text{-}b$ is represented as follows:



Typically, a and b are small, so explicitly representing the structure as a sequence of similar elements does not burden the computer memory too heavily. If b is omitted, and thus codes infinite repetition, the above scheme cannot be used, since one cannot go on infinitely creating new alternatives. However, the infinite repetition can be coded by a self-reference pointing back, so that for the matching routine it seems as if the pattern continues endlessly:

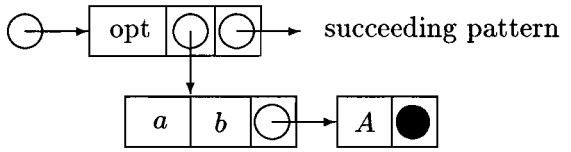


If the minimum number of times the pattern must be present is zero ($a = 0$), this undergoes a slight change:



These representations effectively code infinity, and therefore a terminating mechanism should be provided for when the patterns are matched outside the range of the input. On the other hand, in practical situations this mechanism will seldom be used; either at some point the succeeding pattern matches (leading to a positive result and terminating the matching process) or the optional structure A does not match (leading to a negative result and also terminating the matching process).

However, inside complemented structures such a safety mechanism cannot be used. As argued in the previous section, inside complemented structures all alternatives must be considered. If no further provisions are taken the matching routine will always encounter a new alternative which it will investigate, as it might add new information. These provisions could be taken, of course, but that would burden the matching routine with extra processing, which then would also be executed when finite structures are used. This would deteriorate run-time performance, and therefore a separate internal representation is used in the circumstances that optionality is used inside complementation:

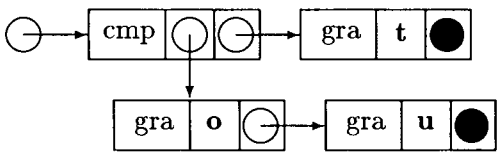


Here, the type field indicates that an optional structure is used, which will cause the matching routine to select the part with the necessary extra processing. The data field contains a pointer to a data item which provides information on the minimum and maximum number of times the optional structure should and may be present. A negative number for the maximum codes infinity. This way of explicitly coding infinity provides the matching routine with additional information, which enables it to determine when the matching process can be terminated.

Complemented structures

The argument of a complemented structure is an arbitrary non-empty pattern. This is the first structure which follows the complementation sign. If one wants to complement a sequence of structures, these should be enclosed in square brackets, ¬[...], thus indicating the range of the complementation (the normal parentheses are already used for optional structures). Confusion with the simultaneity does not occur, since in this case the structure (which is being complemented) only consists of one argument.

The non-empty pattern which is complemented is represented in the same way as it would be, and forms the data field of the complemented structure. For instance, the pattern ¬[**o** , **u**] , **t** is represented as follows:



4.2.3 Summary

With this, all structures and their internal representation have been discussed. It was stated that the internal representation was a modified form of the syntax tree. The goal of the internal representation is to represent the data in such a manner that the matching routine can process them reasonably well and reasonably fast. For that purpose, for instance, the arguments of an alternative structure are linked to the succeeding structure, and the optional structure has been translated to an alternative structure in positive structures. On the other hand, computations which are difficult and laborious to do off-line, such as determining which strings possibly match a complemented structure, are postponed to the run-time part of pattern matching, where they are computationally less complex since only a particular input string has to be matched to the pattern.

The internal representation of patterns is thus a hybrid kind of representation somewhere between a syntax tree and a non-deterministic finite automaton with ϵ-transitions (NFA) (see for instance Hopcroft & Ullman, 1979). The representation for complementation and optionality typically contains structural information of a syntax tree, whereas an alternative structure and concatenations can be viewed as an NFA[1].

[1]The open circles with arrows can be considered as states, the filled circle is the accepting state. The data part of the record, the primitives, are labelled transitions, and the list of structures in the alternative representation are ϵ-transitions.

From the viewpoint of compiler design, this is probably not the most elegant type of representation. The choice for this representation rests mainly on practical grounds. Translating the optional structure to an alternative saves the need to code a separate routine for matching positive optional structures. Linking the succeeding structure to the arguments of an alternative or simultaneous structure saves administrative effort during run-time. On the other hand, computing during run-time whether or not a string matches a complemented structure is probably less complex than transforming the syntax tree of the complementation into some kind of finite automaton. These considerations have resulted in the internal representation to represent patterns as described above.

4.3 The algorithm for pattern matching

We can now consider the matching strategy in more detail. As depicted in Fig. 4.1, the internal representation is one input to the pattern matcher, the user-provided input the other. The pattern matcher is an interpreter which is driven by the organization of the linguistic rules. At the basis patterns are interpreted by the pattern matcher and matched to the input. This is the kernel of the system. In this section the algorithm for matching patterns will be given.

The general task of the matching routine is to determine whether or not a certain pattern—internally represented by the linked list of structures—is present in the input buffers or not. For the present, it is assumed that there is only one input buffer, containing either graphemes or phonemes. In the next section the situation of synchronized buffers is dealt with.

The task of the matching routine is the following. Suppose the rule (4.1) is specified to deal with the pronunciation of words like 'container' and 'trailer'.

$$\mathbf{a,i} \quad \rightarrow \quad \mathbf{EE} \;/\; \neg\mathbf{c} \;\_\_\; \left\{\begin{matrix}\mathbf{l}\\ \mathbf{n}\\ \mathbf{d}\\ \mathbf{e}\end{matrix}\right\}, \left\{\begin{matrix}\mathbf{(d)}, \mathbf{<\text{-}segm>}\\ \mathbf{e}\end{matrix}\right\} \tag{4.1}$$

Suppose the word 'container' is typed, and suppose the focus pattern has been tested and found. Then the internal status will be as follows:

input:	c	o	n	t	a	i	n	e	r	
							↑			
output:	K	O	N	T						

(4.2)

The arrow points at the position where the matching of the right context will commence, directly to the right of the focus pattern. The output buffer has partly been filled, as a result of rules included in the same module, which have already been applied. From the fact that the output buffer is filled to the left of the focus it can be concluded that the module scans its input from left to right.

The task of the matching routine is to match the right context of (4.1) to the internal status of (4.2), and return a boolean-valued result: **true** (it matches) or **false** (it does not match).

Thus, the matching routine can be seen as a function which returns a boolean value and is fed by two parameters: the pattern which to match against the buffer contents, and the position at which to start matching the pattern. The last parameter can actually be subdivided into two parts: the buffer in which to match (momentarily a constant), and the position in the buffer at which to start matching. The combination of buffer and position is called *internal position.*

function *Match*(*patt* : *pattern*;
int_pos : *internal_position*) : *boolean*;

Here, *pattern* is the data structure of the linked list of structures, and *internal_position* a record which contains buffer and position information. Customarily, algorithms will be presented in the above manner; functions and procedures (both starting with a capital letter), variables and types are printed in italics and keywords are printed in bold face.

The functionality of the matching routine can be formalized as follows: *Match*(*patt*, *int_pos*) is **true** iff (if and only if) the pattern *patt* matches a string starting at position *Pos*(*int_pos*) in the buffer *Buf*(*int_pos*). *Pos* and *Buf* respectively select the position and buffer of *int_pos*. Inside this function, we may expect the same differentiation between structures as is present in the syntax and the semantics. In this respect, however, it is relevant to separate the types into two groups: primitives versus 'real' structures. The primitives are characterized by the fact that they refer to exactly one segment in a buffer, whereas the structures do not refer directly to the input buffer but indicate how to combine sub-patterns.

Structures can be concatenated in any order. However, when a structure is used, the semantics prescribes that the full succeeding pattern (the pattern succeeding the structure concerned) must be included in the determination of the meaning of the structure. Therefore, if the function *Match* encounters such

a structure, control will be given to a routine which handles these structures, including their succeeding patterns.

On the other hand, if a primitive is encountered, the matching value of that specific primitive can be determined. If a negative value results, the matching process can be terminated (until a complementation sign is encountered, we are matching a positive structure). If a positive value results, it should continue. Then, both the starting position at which to match and the pattern with which to match must be updated.

Such a recipe can be formulated compactly and explicitly in an algorithm. For this purpose I use a pseudo-Pascal code; Pascal since the program is implemented in that language, and a pseudo variant to be able to leave out irrelevant detail. Thus, the function *Match* can be formalized as follows:

```
function Match(patt     : pattern;
               int_pos : internal_position) : boolean;
begin
    result := true;
    while result and Type(patt) = primitive
    do
        result := Match_primitive(patt, int_pos);
        int_pos := Update(int_pos);
        patt := Select_next(patt)
    od;
    if result and Type(patt) = structure
        then result := Match_structure(patt, int_pos) fi;
    Match := result
end;
```

Type is a function that returns the type field of the current pattern. *Update* updates the internal position. In the current case of one input buffer this consists of shifting the arrow in (4.2) one position to the right or the left, depending on the matching direction. In this case, this is to the right since the right context is being matched, but for left contexts, for instance, this would be to the left. *Select_next* updates the pattern, i.e., selects the next element in the linked list of structures. In true Pascal code this is equivalent to the statement: $patt := patt\uparrow.next$, but to abstract from the implementation this is presented as a function.

Note that as soon as *result* turns **false** the matching process terminates with negative result. On the other hand, if the end of the linked list is reached and *result* is still **true**, the process terminates with positive result; both tests,

Type(*patt*) = *primitive* and *Type*(*patt*) = *structure*, fail (it is assumed that *Type* recognizes the empty list and returns a unique code for that case).

The division between the primitives and structures manifests itself in a call to two different functions, *Match_primitive* and *Match_structure*. These form the topics of the next sections, 4.3.1 and 4.3.2. Amongst other things, *Match_structure* deals with complementation. As will be explained, matching inside complementation differs from matching outside, which is the normal mode. In section 4.3.3 the matching strategy inside complementation is discussed. A brief discussion in section 4.3.4 concludes this section on how patterns are matched against a single input buffer.

4.3.1 Matching primitives

Match_primitive is a boolean function that matches a single primitive to a single segment in a certain buffer. Its functionality is given by: *Match_primitive*(*prim*, *int_pos*) returns **true** iff the primitive pattern *prim* matches the element at the internal position *int_pos*.

Four primitive types can be used: grapheme, phoneme, grapheme features and phoneme features. When a grapheme or phoneme is specified in the rules, that specific grapheme or phoneme must be found in the input buffer. The matching value can be determined with a single statement.

result := *Data*(*patt*) = *Segment*(*int_pos*);

Data returns the contents of the data field, *Segment* selects the segment at the internal position.

When features are specified in the rules, the segment in the input buffer must satisfy all feature specifications. One by one, these specifications are verified, a process that terminates if one specification fails, or when all specifications have been dealt with:

result := *true*;
feat := *Data*(*patt*);
segm := *Segment*(*int_pos*);
while *result* **and** *Present*(*feat*)
do
    result := (*segm* **in** *Feature_set*(*feat*)) = *Value*(*feat*);
    feat := *Select_next*(*feat*)
od;

Present is a boolean function that returns **true** if there are still elements in the linked list to be checked. In this place it is

equivalent to the Pascal statement *feat* $\neq$ *nil*. *Feature_set* is a function that returns the set of all segments which are described by the given feature. *Value* is a boolean function which returns **true** if the specified feature value in the rule is '+', **false** if this is '–'.

The test whether the input-buffer segment is in correspondence with the feature specification consists of two parts. The first test determines whether the current segment is an element of the set denoted by the feature: *segm* **in** *Feature_set*(*feat*). The second test checks if that was intended: "element of feature-set" = *Value*(*feat*). Then, of course, after testing one feature specification, the next one must be selected, which is performed by *Select_next*. If all features have been dealt with, this will cause the test *Present* to fail. Thus, the result will only be positive if the input segment satisfies all the feature specifications. As soon as one of the requirements is not met, the matching process terminates.

The one-input buffer situation which is assumed here does not yet discriminate between graphemes and phonemes. In section 4.4, which deals with the synchronized buffers situation, the distinction will lead to a slightly different algorithm, for which reason the full algorithm for *Match_primitive* will be given there.

4.3.2 Matching structures

As argued in chapter 3, when any of the structures is encountered, the pattern succeeding that structure is necessary to determine the meaning of the structure. For this reason control is transferred to the function *Match_structure* when a structure is encountered. The task of this function is to match the remainder of the input to the remainder of the pattern. The first element of this pattern is a structure.

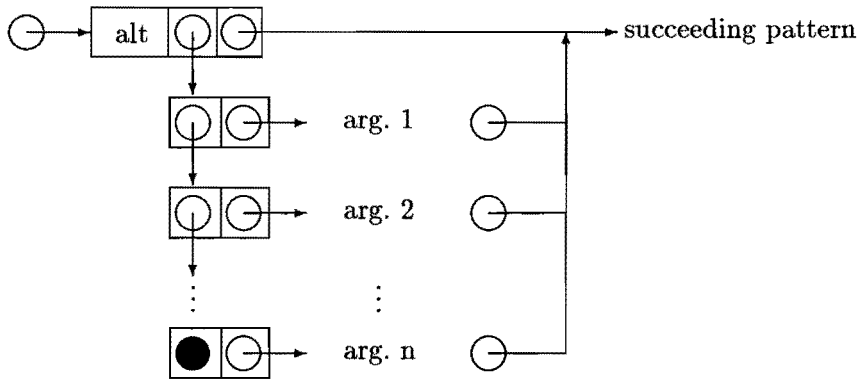
Basically, the matching strategy differs for each structure. Therefore, the only thing the function needs to do is to differentiate between the possible types and transfer control to a function which is specialized to deal with the encountered structure. Since optional structures are coded as alternatives in positive (non-complemented) structures these cannot be encountered, only the alternative, simultaneous and complementation structures must be dealt with:

```
function Match_structure(patt     : pattern;
                         int_pos : internal_position) : boolean;
begin
   case Type(patt) of
   alternative     : result := Match_alt(patt, int_pos);
   simultaneous    : result := Match_sim(patt, int_pos);
   complementation : result := Match_cmp(patt, int_pos)
   end;
   Match_structure := result
end;
```

For each of the three possible types a different function is called. These will be discussed in this order.

Alternative structures

The alternative structure is represented by a list of patterns. The head of each such pattern consists of one of the arguments of the alternative structure, the tail of each argument consists of the succeeding pattern:



Each pattern in the list is exactly like the patterns we encountered at the highest level: they are a linked list of structures. This means that we can use the function we already 'have', *Match*, to determine the matching values of the patterns in the list. Actually, this recursive call follows directly from the semantics, which prescribes how to rearrange the pattern and apply the same function, "the meaning of" to the new patterns.

With this, the strategy for matching an alternative structure becomes simple: *Match* the patterns of the list one by one until one of them matches, and terminate matching with negative result if all patterns fail. Note that

each time a new pattern is tested, it is matched starting at the same internal position as the first time.

```
function Match_alt(patt    : pattern;
                   int_pos : internal_position) : boolean;
begin
    result := false;
    alternative := Data(patt);
    while not result and Present(alternative)
    do
        result := Match(Pattern(alternative), int_pos);
        alternative := Select_next(alternative)
    od;
    Match_alt := result
end;
```

Pattern selects the following pattern from the list of alternatives.

The number of patterns which are represented in the linked list is finite. Either it directly follows from the number of arguments the user has specified, or the alternative structure consists of two elements when it is derived from an optional structure. The routine specified above will therefore always terminate, provided that the internally used routines terminate.

Simultaneous structures

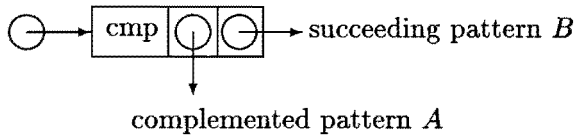
The simultaneous structure can be treated analogously to the alternative structure, with appropriate adjustment of combining the results. Apart from the type indication, the internal representation is the same. Therefore, the simultaneous structure can be treated with the following algorithm:

```
function Match_sim(patt    : pattern;
                   int_pos : internal_position) : boolean;
begin
    result := true;
    simultan := Data(patt);
    while result and Present(simultan)
    do
        result := Match(Pattern(simultan), int_pos);
        simultan := Select_next(simultan)
    od;
    Match_sim := result
end;
```

This routine only terminates with positive result if all of the patterns are found. If one of them fails, the result is negative.

Complemented structures

The complementation structure is represented as a pattern embedded within complementation marks:



To determine whether a string belongs to the pattern, the semantics prescribes that it should be an element of $A^U{\sim}\underline{B}$, but not at the same time be an element of $\underline{A}{\sim}\underline{B}$, where A is the complemented pattern and B the succeeding pattern. If this is implemented straightforwardly, the membership of the input must be established for both sets. There is no simple shortcut, since in chapter 3 it has been shown that in general $A^U{\sim}\underline{B} \setminus \underline{A}{\sim}\underline{B} \neq (A^U \setminus \underline{A}){\sim}\underline{B}$ (the lefthand side is the semi-compositional definition, the righthand side the compositional one). So this would mean more or less a redoubling of the amount of work, as compared to positive structures. Therefore, a more efficient implementation has been searched for.

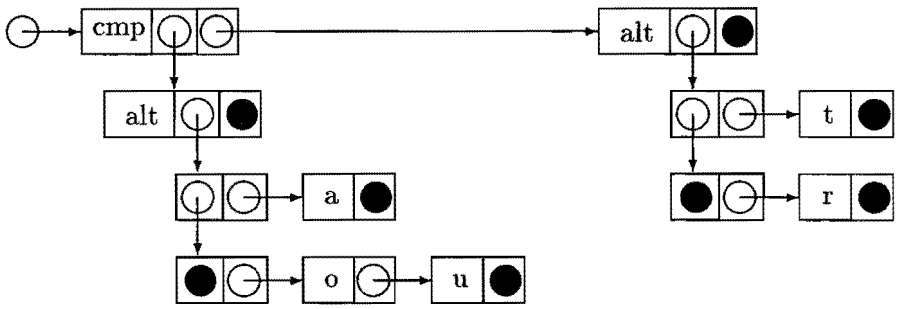
The definition of complementation is derived from a formulation which reflects its origin (see chapter 3, page 55):

$$\underline{\neg A\,,B} = \big((A^U \setminus \underline{A}){\sim}\underline{B}\big) \;\setminus\; \underline{A}{\sim}\underline{B} \tag{4.3}$$

Thus, complementation can be interpreted as follows. $\underline{\neg A\,,B}$ denotes those strings which are *candidates*, but which at the same time are not *explicit nofits*. Candidates are defined as the strings that can be fitted to the pattern such that the segments fitted to the complemented part (A) do not match and those to the succeeding pattern (B) do $\big(=(A^U \setminus \underline{A}){\sim}\underline{B}\big)$. Explicit nofits are defined as the strings that can be fitted such that their segments both match the complemented part and the succeeding pattern ($=\underline{A}{\sim}\underline{B}$). The string 'att' for instance, is both a candidate and an explicit nofit for pattern (4.4) (and should, for that reason, not be included by the set of strings denoted by that pattern).

$$\neg\left\{\begin{matrix}\mathbf{a}\\ \mathbf{o}\,,\mathbf{u}\end{matrix}\right\},\left\{\begin{matrix}\mathbf{t}\\ \mathbf{r}\end{matrix}\right\} \tag{4.4}$$

In patterns we can distinguish paths of concatenated primitives, or briefly: *paths*. For instance, pattern (4.4), the internal representation of which is depicted below, contains four paths: **¬a , t**, **¬a , r**, **¬[o , u] , t** and **¬[o , u] , r**.



A certain string can be a candidate for a specific path, it can be an explicit nofit, or it can be neither, but it cannot be both at the same time for the same path. If a string is both a candidate and an explicit nofit for the pattern, this is due to different paths. Now we can separate the 'matching value' of a string for a certain path into two parts: one for the complemented part, and one for the succeeding pattern. In this way we have a tuple of two boolean values, and thus four possible combinations: (**false-false**), (**false-true**), (**true-false**) and (**true-true**). Here, the value is not yet adjusted for whether or not being complemented, so (**true-true**) means the first part matches the complemented structure and the second part matches the succeeding pattern.

The definition of complementation, as given in (4.3), can now be interpreted as follows: the input string must result in a (**false-true**) tuple for a certain path (it must be a candidate), but it may not result in a (**true-true**) tuple for any other path (it may not be an explicit nofit).

Therefore, the general strategy could be: compute and consider all tuples. If there is a (**true-true**) tuple, the pattern as a whole does not fit, as this is an explicit nofit. If there is no (**true-true**) tuple, then look for a (**false-true**) tuple. If such a tuple is found a positive result must be returned, as a candidate is found which is not an explicit nofit. If such a (**false-true**) tuple is not found, the routine must return a negative result; the string is neither an explicit nofit nor a candidate.

In this scheme, however, the matching value of the succeeding pattern must be determined for each path in the complemented pattern. If the position at which to start matching is different for each path, this has to be done anyhow. For instance, in pattern (4.4), $\{^{\mathbf{t}}_{\mathbf{r}}\}$ (the succeeding structure) must be matched to the second character (relative to the starting position of the whole pattern)

for the first path, $\neg$**a**, and to the third character for the second path, $\neg$[**o** , **u**].

On the other hand, if certain paths of a complemented pattern are of equal length, such as in pattern $\neg\{\begin{matrix}\mathbf{a}\\\mathbf{e}\end{matrix}\}$, **t** the starting position for the succeeding pattern is the same, and therefore the matching values of those cases will all be similar. So in these cases some increase in efficiency can be gained.

This can be done by combining the results of those complemented paths which have equal lengths. For paths within an alternative structure the combining operator is the logical or: if one of them matches, this leads to an explicit nofit if the succeeding pattern also matches. Analogously, for paths within simultaneity this is the logical and.

In fact, it is not the length of the complemented pattern that is essential, but the position where the matching of the succeeding pattern should commence. Therefore, the results of all paths which lead to the same internal position are combined. Thus, we obtain a list of matching values for the complemented part, of which each element is associated with a different internal position. Now, for each list element the matching value of the succeeding structure is determined, and the same strategy for combining result-tuples as before can be applied to determine the matching value for the pattern as a whole.

Yet, there is still some efficiency to be gained. If the list of matching values for the complemented part is ordered in such a way that first for all **true**-values the succeeding pattern is matched, and next for all **false**-values, we can terminate matching the succeeding pattern as soon as a positive result is returned. If the accompanying complemented path is **true** the string appears to be an explicit nofit and thus a negative result can be returned directly. If the accompanying complemented path is **false**, the string appears to be a candidate. But since **true**-values in the result list precede the **false** values, explicit nofits can no longer be encountered. Therefore, a positive result can be returned directly. If the succeeding structure fails to match for all list elements, again a negative result for the whole pattern must be returned. Note that only in this case does the succeeding pattern have to be matched for all list elements. This strategy can be formalized in the following algorithm:

```
function Match_cmp(patt     : pattern;
                   int_pos : internal_position) : boolean;
begin
    res_list := Exh_match(Data(patt), int_pos);
    result := false;
    succ_patt := Select_next(patt);
    while not result and Present(res_list)
    do
```

```
        neg_res := Res_value(res_list);
        result := Match(succ_patt, Int_pos(res_list));
        res_list := Select_next(res_list)
    od;
    Match_cmp := result and not neg_res
end;
```

Res_value select the matching value of the current element of a result list, *Int_pos* selects the internal position from that element.

Exh_match determines for each patch in the complemented pattern its matching value and the resulting internal position. It combines the results for paths with the same resulting internal position, and it orders the results in a list such that those paths that match are included first, and returns this list. In the next section, (4.3.3), this function is discussed in more detail. While no definite decision can be made, i.e., when *result* = **false**, which means neither an explicit nofit nor a candidate has been found, the succeeding pattern is matched at the internal position of the next element of the result list. Storing the accompanying result of each result list element in *neg_res* enables a quick computation of the final result. This can be done with the single statement:

Match_cmp := *result* **and not** *neg_res*

If the succeeding pattern matches for a certain path, then the **while**-loop terminates with *result* = **true**. The final result should now become the inverted value of the accompanying complemented part, which is stored in *neg_res*; if *neg_res* = **true** then an explicit nofit has been found, and thus the final result is negative, if *neg_res* = **false** this indicates a candidate, and thus the final result is positive. On the other hand, if the succeeding pattern did not match for any of the paths (no candidates are found), the loop eventually terminates with *result* = **false**, in which case a negative result is returned.

4.3.3 Exhaustive matching

The function *Exh_match* deserves some additional discussion. It performs the task inside complemented structures that is performed by *Match* outside (in positive structures). Compared to *Match*, it differs in some aspects. First of all, it does not return the matching value of a pattern, but it returns a list of matching values for all paths inside the complementation. Second, for each matching value it includes information on where to start matching the succeeding pattern. Third, it only terminates matching at the end of the complemented part, rather than as soon as *result* turns false.

Despite these differences, there is quite some similarity in task. We will find the same differentiation in type of structure as we saw in *Match*. Also,

more or less similar algorithms are encountered. Again, these algorithms will differ from the previously presented ones in the respects given above. Since for each path an indication is needed on where to commence matching the succeeding pattern, we cannot simply terminate matching a path and return a negative result when a certain primitive appears not to be present. One solution for this is to continue matching until the end of the complemented pattern is encountered. This only affects the termination criterion for the routine. Another solution is to only count the remaining elements of the path, but this has more consequences for the algorithm; more or less a redoubling of source code is necessary, since the structures that can be encountered are the same, only the matching of the primitives differs.

As in practical situations complementation is not used with high frequency, and, moreover, the patterns being complemented are fairly simple (mostly consisting of one or two paths of one or two characters), one may not expect a very spectacular increase of run-time performance when the second solution is chosen in favour of the first. Therefore, in the implementation I chose the first solution. For this reason the routine is called *Exh_match*, as it exhaustively matches each path to its end.

With this, the important differences of *Exh_match* with its positive counterpart *Match* have been discussed, except the fact that optionality must also be dealt with. This structure has not yet been discussed, as it does not occur in positive structures. Therefore, how to deal with this structure and where this is done are discussed in the next section. The algorithms for matching the other structures (alternation, simultaneity and complementation) inside a complemented structure are given in Appendix 4.A.

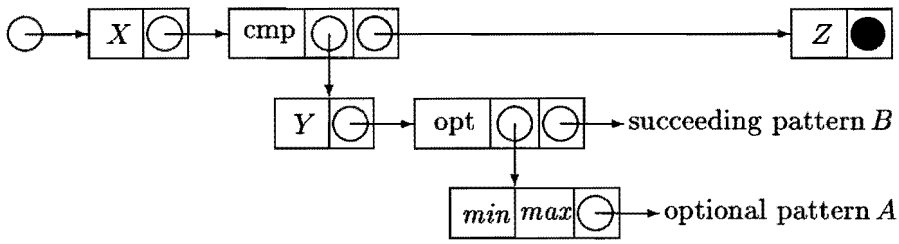
Optional Structures

The only case in which a separate internal representation for optional structures can occur is when optionality is used inside complementation. The general appearance of such a pattern is as follows:

$$X\,,\neg[Y\,,(A)min\text{-}max\,,B]\,,Z \tag{4.5}$$

Here, A, B, X, Y and Z are arbitrary patterns, all of which may be absent, except A. A is also called the optional pattern and B is the pattern succeeding the optional pattern within the complementation. Within the remainder of this section (4.3.3) B will also be called in short the succeeding pattern, in contrast to the general convention where it means the pattern succeeding the complementation (which is Z, here). A and B are of interest for the routine dealing with optionality, X, Y and Z are dealt with by other routines.

The internal representation of pattern (4.5) is as follows:



The sub-pattern matches if the optional pattern A is present at least *min* times and up to at most *max* times before the succeeding pattern B. Of course, in the end the matching result should be inverted, since the structure is present in a complementation. However, the interpretation of the complemented structure as a whole is dealt with by the routine dealing with complementation, *Match_cmp*. Here, we are concerned with determining the matching values of the paths inside the complemented structure, that is, determining the first values of the result-tuples described in the previous section (4.3.2). Therefore, the task of this routine, which will be called *Exh_match_opt*, and which deals with optionality inside complementation, is to determine whether the optional structure (A) followed by the succeeding pattern (B) is present or not.

A straightforward strategy to determine this is as follows. First match the optional pattern A *min* times and then match the succeeding pattern B. This gives the first matching result to be put out in the result list. Just before B is matched, the current internal position is saved, so that the next option, A^{min+1}, can be checked without having to match the first *min* occurrences once again. Then, after matching A once, B is matched again and the result is added to the result list. Each time a next occurrence of the optional pattern has been matched the internal position is saved. This continues until the optional structure has been matched up to *max* times. The following algorithm does the job:

```
for nr := 1 to min do "match optional pattern A" od;
save := int_pos;
"match succeeding pattern B";
res_list := result;
for nr := min + 1 to max
do
    int_pos := save;
    "match optional pattern A";
    save := int_pos;
    "match succeeding pattern B";
```

res_list := *Unite*(*res_list*, *result*)
od;
return *res_list*;

Unite is a function which simplifies two result lists such that each internal position only occurs once.

As to this algorithm, there are, however, two complicating factors:

- for each matching operation, a list of result values may result, as matching takes place inside complementation. All elements of the list should be dealt with.
- *max* may be infinite, which cannot be dealt with by the routine above.

As to the first factor, we may assume that when we encounter the optional structure, we are investigating a specific path. There can of course be more paths before an optional structure, but for each path the optional structure will be investigated individually. So we start matching the optional structure at a certain internal position, transported to the routine by means of the variable *int_pos*.

It is possible that the optional structure contains an alternative structure, which contains paths of different lengths. The result of matching structure A may therefore be a list of result values. If, in succession, we must once again match A or match the succeeding pattern B, we must do this for each list element. So, generally, on the one hand a list of result values is returned by the matching process (the *result list*), and on the other hand a list of states at which to commence matching is presented to the matching process (the *state list*). Only the first time we start matching we know we have to deal with exactly one state. This can be seen as a state list containing one element.

The result list and the state list are essentially of the same type: each element of a list contains information on where to continue or commence matching (by means of the internal position) and the resulting or initial matching value. The matching value of an arbitrary pattern inside complementation can then be seen as a function of the pattern and the list of commencing values, which results in a list of result values:

result_list := *List_match*(*pattern*, *commence_list*);

Inside the function *List_match* the pattern concerned should be matched for each list element. Each list element contains an internal position and an initial matching value. The internal position is selected by *Int_pos*, the initial matching value by *Res_value*. The initial matching value is needed for a correct result; if the first $n-1$ occurrences of A do not match, then,

despite the fact that the n^{th} occurrence may be present at that particular position, the sequence A^n still does not match. This is taken care of by the function *Adjust*. Each matching process produces a result list, and all the result lists are combined as before, that is, simplified for paths ending at the same internal position by the function *Unite*. This is expressed below:

```
function List_match(patt      : pattern;
                    comm_list : result_list) : result_list;
begin
    res_list := empty;
    while Present(comm_list)
    do
        int_pos := Int_pos(comm_list);
        prv_val := Res_value(comm_list);
        tmp_list := Exh_match(patt, int_pos);
        tmp_list := Adjust(prv_val, tmp_list);
        res_list := Unite(tmp_list, res_list);
        comm_list := Select_next(comm_list)
    od;
    List_match := res_list
end;
```

Now, the routine for matching optional structures, given above in an informal style, can be made more explicit. For non-infinite matching of optional patterns in complemented structures we can give the following routine:

```
function Exh_match_opt(patt      : pattern;
                       comm_list : result_list) : result_list;
begin
    opt_patt := Data(patt);
    succ_patt := Select_next(patt);
    for nr := 1 to min
        do comm_list := List_match(opt_patt, comm_list) od;
    res_list := List_match(succ_patt, comm_list);
    nr := min;
    while nr ≠ max
    do
        nr := nr + 1;
        comm_list := List_match(opt_patt, comm_list);
        res_list := Unite(List_match(succ_patt, comm_list), res_list)
    od;
    Exh_match_opt := res_list
end;
```

Saving the internal positions is now done implicitly in the result lists, which serve as state lists for the next matching process. Note that also the internal position where matching is first to commence is transferred to the function by means of a state list. That state list, *comm_list*, contains, apart from the internal position, also the initial matching value, the result of (one of the paths) of pattern Y.

The last **for**-loop is replaced by a **while**-loop to be able to make the last step: to make the routine suited to deal with 'infinite matching', this is when $max = -1$, and the routine would therefore not terminate. In a practical system the physical string is always finite. This means that the end of that string will always be reached by repetitive matching of the optional structure. This can be detected by *Exh_match_opt* by means of a special function, *Out_of_bounds*. This function returns **true** when all of the position pointers are outside of the input range. This can be added to the termination criterion:

while $nr \neq max$ **and not** *Out_of_bounds*(*comm_list*) **do** ...

Thus the relevant repetitions are dealt with, since the paths of the optional structure which fall outside of the input range will cause the pattern succeeding the complementation (Z) to fall outside of the input range as well, and therefore a candidate can never result for those paths. Therefore, *Exh_match_opt* can be terminated as soon as repetitions reach beyond the input range.

The algorithm for exhaustive matching

Now we can return to the general routine which deals with matching inside complementation, the function *Exh_match*. As stated above, this function resembles the function *Match*, but differs in three respects: it returns a list of matching values rather than a single result, it also returns the accompanying internal position, and only terminates at the end of a pattern (which is the end of the complementation). Thus, the following algorithm results:

```
function Exh_match(patt    : pattern;
                   int_pos : internal_position) : boolean;
begin
    result := true;
    while Type(patt) = prim
    do
        result := Match_primitive(patt, int_pos) and result;
        int_pos := Update(int_pos);
        patt := Select_next(patt)
    od;
```

```
    res_list := Create_list(result, int_pos);
    case Type(patt) of
    alternative      : res_list := Exh_match_alt(patt, res_list);
    simultaneity     : res_list := Exh_match_sim(patt, res_list);
    complementation : res_list := Exh_match_cmp(patt, res_list);
    optionality      : res_list := Exh_match_opt(patt, res_list)
    esac;
    Exh_match := Sort(res_list)
end;
```

The first part is the same as function *Match*, save for the fact that the routine does not terminate when *result* turns **false**. When all initial primitives have been matched, an initial state list is created by *Create_list*. This list serves either the as starting point for the routines which deal with structures, or as a result list when no structures are used and the end of the complementation or pattern has been reached. If a structure is encountered the structure-specific routines are called. The last, *Exh_match_opt* is discussed above. The first three, *Exh_match_alt*, *Exh_match_sim* and *Exh_match_cmp* are the exhaustive matching counterparts for 'positive' routines discussed in section 4.3.2, and are given in Appendix 4.A. When the remainder of the complemented pattern has thus been handled, the last task of the routine is to rearrange the result lists such that those paths that match are in first position, so that the routine *Match_cmp* can terminate as soon as an explicit nofit or a candidate has been found. This is done by the function *Sort*.

4.3.4 Summary

With this, the important characteristics of the matching routine have been discussed, when the input consists of a single buffer containing segments. The matching routine is part of an interpreter, which interprets the linguistic rules one by one. The kernel of the interpreter is the function *Match*, which matches an arbitrary pattern to arbitrary input.

The function *Match* is syntax-directed, that is, the same structure which is found in the syntax of patterns is found in *Match*. Essentially, *Match* is called recursively, just as non-empty patterns can be used recursively. However, due to the implementation of complementation (which avoids redoubling of computational effort), inside complementation a slightly different matching strategy is applied. Therefore, when complementation is used, *Match* is not called recursively, but the function *Exh_match* is called. This function exhaustively matches the complemented pattern, and returns a list of matching values and internal positions, essentially one pair for each path. Inside

Exh_match all recursions of non-empty patterns call on *Exh_match*, so here recursion is restored.

The situation of synchronized buffers, as opposed to a single input buffer, only affects the routine for matching primitives. The routines for matching structures remain unaltered. In fact, the only extra task *Match_primitive* must perform is to determine in which of the synchronized buffers the specified primitive is to be looked for. This is the topic of the next section.

4.4 Synchronized buffers

Synchronized buffers are needed to synchronize the input with the output. The term synchronization is used in imitation of Susan Hertz, who introduced the notion in her Delta system (Hertz, Kadin & Karplus, 1985). Although synchronization is generally used to indicate time alignment between processes, here it will be used to indicate segment alignment between buffers. In TooIJiP synchronization is needed for two purposes. On the one hand it is needed to be able to use information on orthography and pronunciation simultaneously in the rules. This will be called *internal synchronization*, since it does not necessarily manifest itself on the level of original input and final output. On the other hand it is needed to determine overall input-to-output relations, that is, how the characters of the original input correspond to those of the final output. This will be called *overall synchronization*. Both overall synchronization and the possibility to use information on orthography and pronunciation are special features which were required in the design of TooIJiP (see chapter 2).

Synchronization of buffers, or, more in general, synchronization of two or more levels means that each segment or sequence of segments at one level can be associated with a (sequence of) segments at another level. For instance, the orthography and pronunciation of the word 'cadeau' /KAADOO/ (present) are associated as follows:

input:	c	a	d	e a u	
output:	K	AA	D	OO	

(4.6)

In words: the letter 'c' is pronounced as a /K/, the 'a' as an /AA/, the 'd' as a /D/ and the sequence 'eau' as an /OO/.

For retaining the input-to-output relations each unit which converts input to output (in TooIJiP these are modules) must have separate buffers for input and output. During the conversion process these buffers are synchronized according to the rules that apply. For instance, in (4.6) apparently a rule of the form "**c** → **K** / some context" has been applied. Apart from adding the

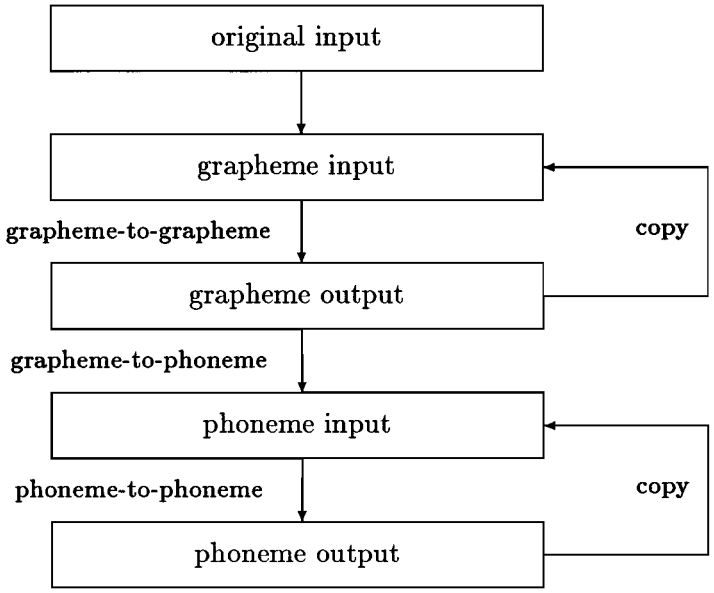


Figure 4.2: Buffer architecture of TooIjP.

'K' to the output, the system will also synchronize the 'K' in the output with the 'c' in the input.

To achieve overall synchronization, that is between the input and the result of the consecutive modules, one can create a buffer for the initial input and a separate buffer for the output of each module, which simultaneously serves as input buffer for the next module. Synchronization between each two subsequent buffers then directly determines the overall synchronization. Thus one needs $n + 1$ buffers where n is the number of modules.

There is a way, however, to make the number of buffers fixed and independent of the number of modules. As explained in chapter 2, there are three types of modules in TooIjP: one which manipulates graphemes, one which transforms graphemes into phonemes, and one which manipulates phonemes. Instead of creating an output buffer for each module, it is also possible to create an output buffer for each type of module. Then, four buffers are needed for the conversion process: one for the input of the first type of module, and three for the output of each type of module. In this scheme the contents of the output buffer are copied to the input buffer in the case of successive modules of the same type. For overall synchronization one extra input buffer is needed to preserve the original input (see Fig. 4.2); for if two or more grapheme-to-grapheme modules are used in the four-buffer scheme, the original input is overwritten by the output of the first module.

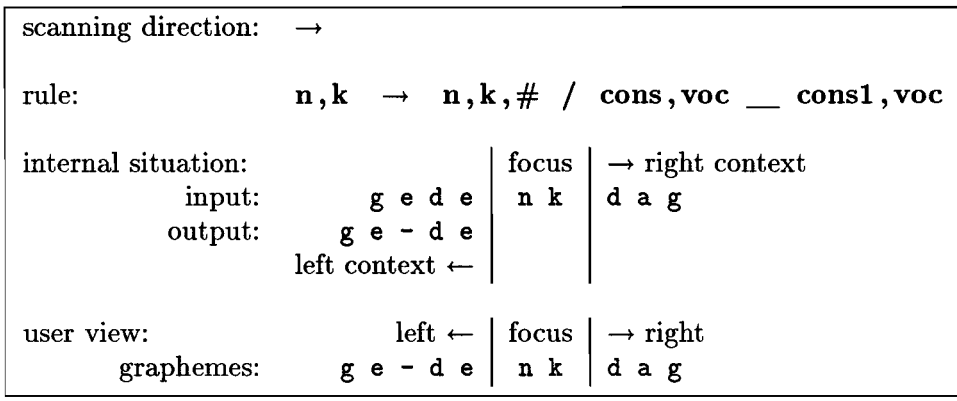


Figure 4.3: Selecting buffers in GTG modules.

In TooJiP the scheme of five data buffers is implemented. In section 4.4.1 it is discussed how the situation of five synchronized buffers affects the routine *Match_primitive*. In section 4.4.2 two possible mechanisms to implement synchronization are discussed, which are compared in section 4.4.3. The implementation of the one that is to be preferred is discussed in section 4.4.4.

4.4.1 Matching primitives to synchronized buffers

In section 4.3.1 a strategy for matching primitives is presented for the case of a single input buffer. There, it is assumed by pre-condition that the buffer in which to match and the position at which to match are known. Each time a primitive is matched, afterwards the new matching position is computed, thus satisfying the pre-condition for the next time a primitive will be matched.

The actual situation with synchronized buffers is slightly different. There are three situations to be distinguished, corresponding to the three types of modules. In the grapheme-manipulating modules so-called grapheme-to-grapheme (GTG) rules are used. One may only refer to graphemes, since at that point no phonemes are yet available. When graphemes are referred to, either the grapheme input buffer or the grapheme output buffer is consulted. This depends on the pattern which is being matched and the direction in which the module's input is scanned. If the input string is scanned from left to right (→) then the left context will be matched against the output buffer, and the right against the input buffer (see Fig. 4.3). Conversely, if the input string is scanned from right to left (←), the buffers against which the patterns are matched are exchanged accordingly. The focus pattern, of course, is always matched against the input buffer. In this way, it seems to the user as if there is only one buffer, where all transformations are executed

scanning direction: →

rule: **a,u → OO / SJ __**

internal situation =

user view:	graphemes left ←	focus	→ right context
input:	c h	a u	f f e u r
output:	SJ		
	phonemes left ←		

Figure 4.4: Selecting buffers in GTP modules.

scanning direction: ←

rule: **O → <*1stress*> / { g,a / e,n,v,e },l __ P**

internal situation:	graphemes left ←	focus	→ graphemes right
graphemes:	e n v e l	o	p j e
input:	E N V C L	O	P J C
	phonemes left ←	focus	
output:			P J C
			→ phonemes right

user view:	graphemes left ←	focus	→ graphemes right
graphemes:	e | n | v | e | l	o	p j | e
phonemes:	E | N | V | C | L	O	PJ | C
	phonemes left ←	focus	→ phonemes right

Figure 4.5: Selecting buffers in PTP modules.

immediately.

In the second type of module, grapheme-to-phoneme (GTP) rules are used. These transform graphemes in the input buffer to phonemes in the output buffer. Therefore, when graphemes are referred to, these should always be searched for in the input buffer. Phonemes, on the other hand, are searched for in the output buffer (see Fig. 4.4). However, one may only refer to phonemes in contexts where they are present, so for left-to-right scanning this is the left context and for right-to-left scanning the right context. If one refers to a phoneme in the other context, a compile-time error occurs.

In the phoneme-manipulating modules phoneme-to-phoneme (PTP) rules

are used. Since derivational history is being retained, one may refer to graphemes as well as phonemes, or require that a phoneme is derived from a (sequence of) graphemes. Reference to phonemes in these modules works the same as reference to graphemes in the GTG modules, that is, phonemes are searched for in the output buffer if that already contains information, and otherwise in the input buffer. However, when graphemes are referred to in PTP rules, these must be searched for in the grapheme level. In this case the last grapheme level preceding the phoneme level is used, the grapheme output buffer, in which the results of the last GTG module are stored (Fig. 4.5).

Particularly in the last two types of modules, when graphemes and phonemes may both be referred to, the buffer to which the primitive should be matched depends upon which of the two is used. This means that the pre-condition of the original algorithm, having selected the internal position, cannot be satisfied without knowledge of the type of the primitive that is used.

Therefore, a different strategy is more appropriate: select and adjust the internal position just before matching the current primitive, rather than prepare it afterwards for the next matching action. In other words, the new pre-condition is that the internal position indicates the position where the previous primitive has been matched. This new condition on the one hand implies that the internal position should be initialized properly before the first matching action takes place, but on the other hand, it is not prepared in vain after the last matching action.

For the algorithms presented thus far, this has some consequences. The selection of the position—previously done in *Match*—will now be part of *Match_primitive*, to be combined with the selection of the buffer. As this depends on the type of primitive (grapheme or phoneme), inside this routine differentiation between these cases must be made. Therefore, as to matching primitives, the following routines perform the tasks.

```
function Match(patt     : pattern;
               int_pos : internal_position) : boolean;
begin
    result := true;
    while result and Type(patt) = primitive
    do
        (result, int_pos) := Match_primitive(patt, int_pos);
        patt := Select_next(patt)
    od;
    if result and Type(patt) = structure
        then result := Match_structure(patt, int_pos) fi;
```

```
    Match := result
end;

function Match_primitive(patt     : pattern;
                         int_pos : internal_position)
                                 : (boolean, internal_position);
begin {pre: buffer and pos pointing at last matched segment}
    (int_pos, fail) := Select_int_pos(Type(patt), int_pos);
    if fail then result := false
    else
        segm := Segment(int_pos);
        case Type(patt) of
        grapheme, phoneme :
            result := Data(patt) = segm;
        g_feat, p_feat : begin
            result := true;
            feat := Data(patt);
            while result and Present(feat)
            do
                result := (segm in Feature_set(feat)) = Value(feat);
                feat := Select_next(feat)
            od   end
        esac
    fi;
    Match_primitive := (result, int_pos)
end;
```

The major part of the routines is simply copied from the previous algorithms. New in *Match_primitive* is the function *Select_int_pos*, which determines the new internal position. The first parameter indicates the type of the pattern to be matched, the second is the current internal position, the value of which has to be adjusted. The variable *fail* indicates whether the returned values are valid; the position parameter might have been moved outside the buffer boundaries, or at the current position there might not be synchronization with the desired buffer. Note that this construction serves as a termination criterion when infinite repetition is used.

4.4.2 Synchronization mechanisms

The task of routine *Select_int_pos* is to determine the new internal position. If this happens to be in the same buffer this is easy, just increment or decrement

pos by one. If, on the other hand, it turns out to be in a different buffer, one must first find the position in that buffer which is aligned with the original position in the original buffer. For this it is important to know how the synchronization mechanism works.

One can probably think of several mechanisms to synchronize buffers. Two of them are prompted directly from the two different purposes for which synchronization is needed in TooJiP. In this section I will discuss these two mechanisms, how they relate to each other, how—by means of synchronization—one can switch from one buffer to another, and what is needed to have synchronization operate correctly.

Two mechanisms

The first mechanism stems from the need to select an adjacent buffer if both graphemes and phonemes are referred to in one pattern. The idea is to switch from the current position in the current buffer directly to the correct position in the adjacent buffer by means of a direct synchronization between the segments of each buffer. This synchronization mechanism will therefore be called *direct buffer switching* (DBS). An integer is attached to each segment in a buffer, which points to the segment in the adjacent buffer with which it is synchronized. In this way, individual segments can be synchronized with values that fall within the buffer range.

Apart from synchronization of individual segments, it must also be possible to synchronize a sequence of segments as an inseparable unit to a segment or another sequence. For instance, the 'eau' sequence of the word 'cadeau' is pronounced as a single phoneme /OO/, which one would like to represent in the system as the grapheme sequence 'eau' being synchronized with the phoneme segment 'OO'. Also, it must be possible to represent insertions or deletions, that is, when a segment in one buffer is not associated with any segment in another buffer. These two cases of alignment can be represented by pointer values that fall outside the buffer range. For synchronization purposes insertions and deletions are the same, and can therefore be represented by a single value. This value will be denoted by a '=' sign. Inseparable sequences will be represented by a different value, which will be denoted by a '-' sign.

To synchronize two buffers in the DBS mechanism, the leftmost segment of an inseparable sequence (which can consist of a single segment) receives a pointer value. This is either a normal value (inside the buffer range), or the insertion value '=' to denote an insertion or deletion. The other segments of the sequence receive the 'inseparable' value. For instance, the synchronization of the orthography of 'cadeau' with the pronunciation of 'KADOO' is as follows (compare with (4.6)):

gra	c	a	d	e	a	u
next:	1	2	3	4	-	-
prev:	1	2	3	4		
phon	K	AA	D	OO		

The 'c' is associated with the first element of the next buffer, the 'K', which in turn is associated with the first element of the previous buffer. Similarly, the 'a' is synchronized with the 'AA', the 'd' with the 'D' and the sequence 'eau' with the 'OO'.

The second mechanism stems from the need to determine overall input-to-output relations, and is inspired by the synchronization method used by Susan Hertz *et al.* (1985) in her Delta System. This mechanism is called *overall synchronization* (OS). Rather than directly synchronizing segments with each other, the positions between the segments are synchronized. Before and behind each segment one or more synchronization marks are placed, and buffers are synchronized at a certain place if the values of the synchronization marks are the same. Thus, synchronization is defined between any number of buffers, rather than between two adjacent buffers. This mechanism can for instance be implemented by attaching a (list of) synchronization marks (sync marks) to each segment, which represents the sync marks behind the segment. In front of the first segment a similar list is placed.

'Normal' one-to-one synchronization of segments is characterized by the fact that sync marks enclosing the segments have the same value. Inseparable sequences are characterized by the fact that certain sync marks are present in one buffer but absent in another. Insertions and deletions are characterized by the fact that between certain segments more than one sync mark is present. With this mechanism the synchronization for 'cadeau', for instance, is as follows:

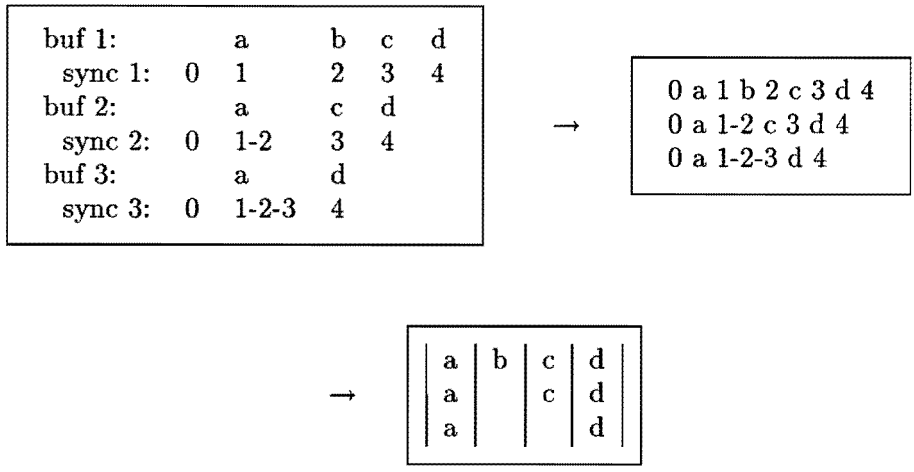
gra		c	a	d	e	a	u
sync:	0	3	7	6	8	1	5
phon		K	AA	D	OO		
sync:	0	3	7	6	5		

The values of the sync marks are arbitrary; of importance is only whether the values are the same. Here, the grapheme 'c' and the phoneme 'K' are aligned as they are 'enclosed' between sync marks with the same values

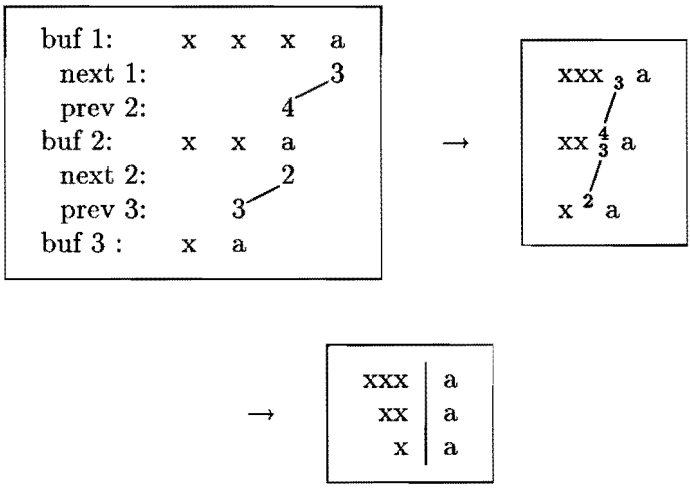
(0 and 3). The same goes for `a-AA` and `d-D`. Since there are no sync marks with the values 8 and 1 in the phoneme sync buffer the grapheme sequence '`eau`' is associated with the phoneme '`OO`'.

Equivalence

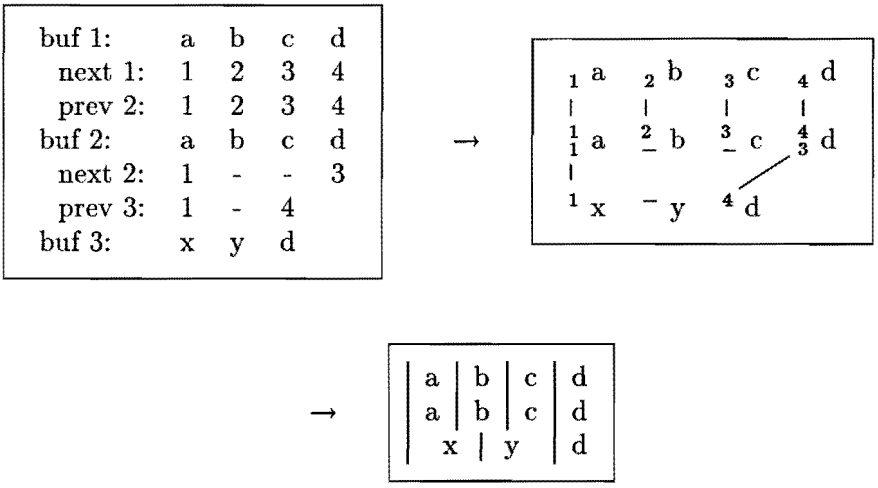
The two synchronization mechanisms are equivalent in the sense that they both describe synchronization fully and can be transformed into each other. Taking a broader viewpoint than only the scheme of grapheme and phoneme buffers, both synchronization mechanisms can be transformed into the already implicitly introduced scheme of vertical lines (see for instance (4.6)) for an arbitrary number of buffers. The OS mechanism is a direct implementation of this representation: the sync mark values can be placed between the segments, and the marks of equal values can be connected with ragged lines, which, so to speak, are then pulled straight:



The correspondence of the DBS method to the vertical line representation is more complex. A normal pointer value, that is, a value which falls within the buffer range, should be interpreted as a sync mark to the left of the character to which it belongs; the segment which is pointed to has an associated pointer pointing back. This is called *consistency* of DBS synchronization. Thus, the two adjacent buffers are synchronized between those two segments. Such synchronization can be extended to other buffers if they, too, at that position have such a synchronization:

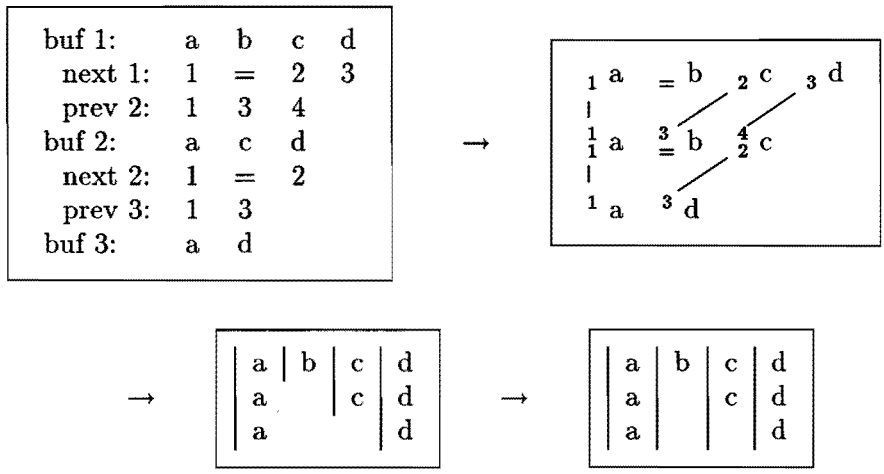


Inseparable sequences are characterized by the fact that they do not have left-synchronization to one of the adjacent buffers (but perhaps do to the other one):

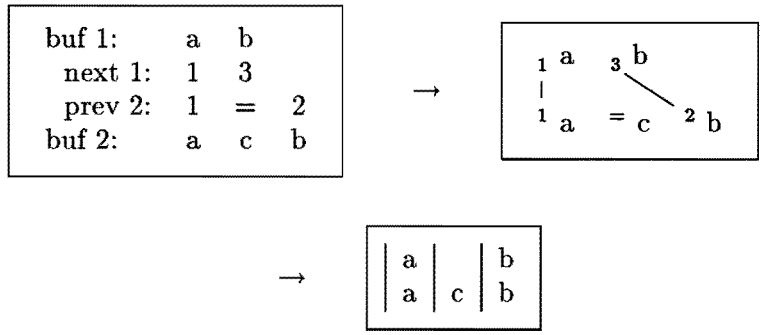


Deletions and insertions can only be detected in the buffer where they occur, since in the relevant adjacent buffer there is no character to be associated with them. Here the synchronization lines have to "propagate" through the buffers; deletions propagate forwards and insertions backwards. Deletion and insertion lines stick, so to speak, to the closest synchronization line to their right.

Deletion:



Insertion:

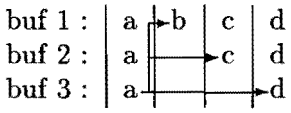


Via the vertical line representation the two synchronization mechanisms can be tranformed into each other and hence they are equivalent. The vertical line representation is the most natural way to represent synchronization, and therefore in TooJiP it is the actual user interface. On the one hand, when synchronization information is given, this is done in the vertical line representation. On the other hand, the concatenation comma and the square brackets the user uses in the rules can be mapped directly on synchronization marks.

Buffer switching

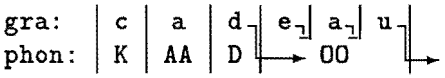
Synchronization between buffers is defined by means of the vertical line representation: one may switch to another buffer along the synchronization marks. So for instance, when the internal status of the system is as depicted below

in the vertical line representation, starting at the 'a' in the third buffer and moving one position to the right (this is typically needed when patterns are matched; after testing one segment an adjacent one must be tested), one can shift to the first buffer and reach the 'b', or shift to the second buffer and reach the 'c', or stay in the third buffer and reach the 'd':



To state it in general: between each two segments a list of sync marks is present. The list always contains at least one element. The first element that is encountered is the synchronization point (that is, the leftmost sync mark when scanning →, or the rightmost sync mark when scanning ←); all buffers which contain the same sync mark (that is: with equal sync values) are synchronized at that position. Thus, in principle, one can switch to any of those buffers. In that respect, starting at the 'a' in the third buffer once again and moving →, one cannot end up in the 'c' in the first buffer by taking the second sync mark between the 'a' and the 'd' in the third buffer.

Also, inside an inseparable sequence one cannot shift to the adjacent buffer which 'caused' the sequence to be inseparable. For instance, moving to the right, one cannot shift from the 'e' in the grapheme buffer to the 'OO' in the phoneme buffer:



What is possible, is to move from the grapheme 'd' to the phoneme 'OO', or from the grapheme 'u' to the phoneme following the 'OO', or to require alignment of 'eau' with 'OO'.

Installing synchronization

Since synchronization is introduced for two purposes, it must meet the two requirements of being able to switch from one buffer to another and to determine the overall input-to-output relationships. For both mechanisms we will now look at how this influences the installation of synchronization, that is, which actions must be taken to fulfil the two requirements.

When a rule applies, that is, when all three patterns of focus, left and right context match, the structural change is added to the output and synchronized with the focus part of the input. Thus, one or more segments in one buffer

are synchronized with zero or more segments in another buffer. The case of zero segments corresponds with an insertion or deletion, the case of one segment with a 'normal' synchronization, the case of more than one segment corresponds with an inseparable sequence.

In the OS method the structural change which is added to the output is synchronized with the input segment or sequence at the outer ends: the sync marks to the left and right of the structural change are synchronized with (given values equal to) the sync marks of the input buffer, which are defined in a previous stage. If the structural change consists of a single segment no further action is necessary. Between the segments of an inseparable sequence new sync marks are generated in the output buffer.

If a rule deletes one or more characters, no character is added to the output, but at that place the output buffer is synchronized with the input buffer at two places, thus representing the deletion. Thus, a list of sync marks results in the output, the head of which is (the list of) sync marks to the left of the deleted segment(s), and the tail of which is the synchronization to the right of the segment(s). Such a list will propagate forwards through the buffers automatically:

buf 1:	a	b	c	d	**b → 0**
buf 2:	a		c	d	**c → 0**
buf 3:	a			d	**d → e**
buf 4:	a			e	

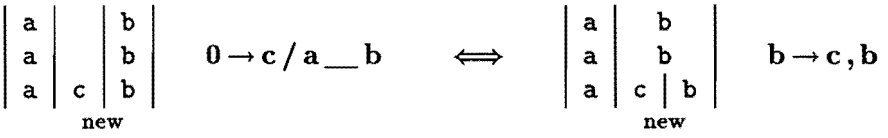
Insertions, however, are a little different, for in the input buffer left and right sync marks are not both available; only one side (depending on the scanning direction) is available. For instance, when the rule

$$\mathbf{0} \rightarrow \mathbf{c} \;/\; \mathbf{a} \_\_ \mathbf{b} \tag{4.7}$$

is specified, and the input buffer consists of:

| a | b |

the character 'c', which is added to the output, only has the sync mark between the 'a' and the 'b' to refer to. Therefore, a new sync mark is introduced at the outer side of the insertion, that is, at the right side in forward scanning, and at the left in backward scanning. This sync mark must propagate backwards through the existing buffers to distinguish the case from an inseparable sequence. Compare, for instance:



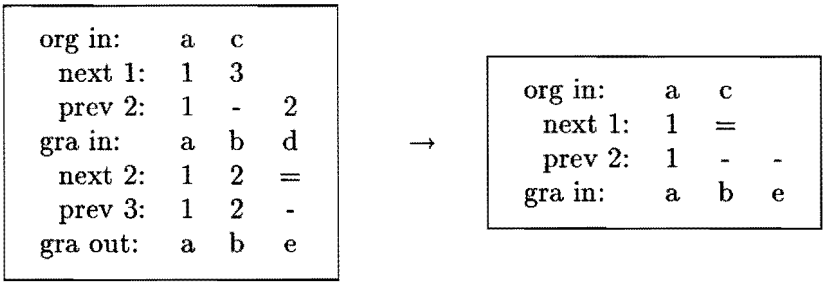
The difference between these two cases is that in the insertion case one can switch to a previous buffer along the new sync mark, and in the inseparable sequence case one cannot.

With insertions propagating backwards, the OS mechanism both ensures correct buffer switching and overall synchronization, irrespective of the number of buffers which are actually implemented. The only demand is that insertions propagate back in previous buffers as far as they can.

Since the DBS mechanism is equivalent to the OS mechanism, DBS automatically also meets these two requirements. However, the preassumption for this equivalence is that the basic synchronization of adjacent buffers is consistent, i.e., that pointers are always mutually pointing at each other. If each module has its own output buffer, this condition is automatically satisfied, for when output is added to the output buffer it is synchronized consistently, and no changes to this information are made further along the derivation. However, if the five-buffer scheme is used (see Fig. 4.2), this is not automatically the case. Suppose, for instance, that a GTG module has been applied and that the output is synchronized with the input. The synchronization of the original input with the output of this module is only defined via the grapheme input buffer, and hence one may not delete the synchronization information by overwriting it with the information of the output buffer.

Therefore, both the pointers of the original input pointing at the input buffer and the pointers of the input buffer pointing backwards will have to be recomputed so as to make the synchronization consistent. Rather than going into the details of this algorithm, I will give an example of how these buffers are adjusted.

Consider the situation depicted below:



Apparently the 'a' in the original input has expanded to the sequence 'ab' in previous modules. The 'c' has been changed into a 'd'. In the current module, the 'b' expands to 'be', and the 'd' is deleted. Considering the overall synchronization—which at that moment is the synchronization of original input and grapheme output—it is as if the 'a' has expanded via 'ab' to 'abe', and the 'c' has been deleted (via 'd'). When the grapheme output is copied into the grapheme input buffer, the synchronization pointers should be adjusted such that they code the cumulative effects of all previous modules, as is depicted above.

Summarizing, it can be stated that both mechanisms need some special attention during the conversion process to ensure that they are reliable. For the OS method insertions must propagate backwards through existing buffers, for the DBS method resynchronization is needed when an output buffer is copied to the input buffer.

4.4.3 Comparison of the two mechanisms

To be able to use synchronization for the two intended purposes, four functions must be implemented in the system. The first one installs the synchronization, that is how the segments in the output are associated with those in the input. The second function is the mechanism to switch from one buffer to another. A third function is to keep the synchronization consistent when the contents of the output buffer are shifted to the input buffer when the next module is to be consulted. The last function determines the overall synchronization from input to output, that is, from the original input grapheme to the final phoneme output.

In particular the last two functions are sensitive to the mechanism used. In the OS method, they are more or less trivial, since the sync marks directly code the desired information. In the DBS method, however, considerable computing effort is needed, since the information is coded indirectly, that is, only between the adjacent buffers synchronization information is available, which will have to be made more global, in a manner described above.

On the other hand, one may expect the DBS method to be more appropriate for the second function, since the information to switch between two adjacent buffers is coded directly by means of the pointers, except for the rare occurrences of an insertion or deletion. Most commonly it is desired to switch between two adjacent buffers, and otherwise there is only one intermediate buffer, in which case a second buffer switch is needed. In the OS method the synchronization will have to be searched for, a process one may expect to be slower than direct switching.

The first function is more or less of the same complexity for both mechanisms. In both cases the sync marks have to be installed according to the mechanisms scheme, which in both cases comprises comparable action.

The order of magnitude in which these functions are used varies greatly. The first function, installation of sync marks, is needed in each module in the order of magnitude of the number of segments to be transcribed by the module. The need for buffer-switching (the second function) depends greatly on the kind of rules being used. In PTP modules this may be a number of times within one rule, but it may also be absent. The third function, keeping the database consistent is needed less than the number of modules per overall transcription, and the fourth function is needed only if one is interested in the derivational history, and in that case once per transcription.

It is hard to tell on theoretical grounds how the balance will dip. Therefore, the two have been implemented as alternatives, and have been tested on a practical situation. The pronunciation of Dutch words was determined by some 8 modules, where buffer switching was used frequently in only one module which contained some 80 rules. It turned out that the two mechanisms were more or less in balance, that is, the DBS method was some 5 % faster than the OS method.

However, I favour the OS method above the DBS method. The main reason for this is that it directly relates to the vertical-line representation. And in spite of the fact that the DBS method originated from the idea of directly switching to adjacent buffers, synchronization and buffer switching are defined in terms of the vertical line representation. In the common situation of one-to-one synchronization of input-to-output segments the DBS method is undoubtedly faster, but when inseparable sequences and in particular insertions and deletions occur, in one way or another the DBS representation must locally be transformed to the vertical line representation before a correct buffer switch can be made. This also means that the designing of buffer switching algorithm is simpler for the OS method than for the DBS method. For a complex system in development, simplicity of such a mechanism, which influences several parts of the system, was thought to compensate a 5 % loss of run-time efficiency. Moreover, when the system needs to be extended in the sense that extra layers of information must be synchronized with existing ones, the OS method is superior to the DBS method. With the OS method the extra layers are the same as any other buffer, with the DBS method the original buffers have to be extended with extra pointer buffers, and some of the algorithms have to be extended, too. For comparison both mechanisms have been implemented, but for the above reasons the latter mechanism, the OS method, is preferable and has therefore been chosen to serve as the synchronization mechanism both for TooIJP in the released versions and TooIJP

in future development.

4.4.4 The algorithm for buffer switching

At this point we can return to the algorithm and see how it works out in the function *Select_int_pos*. The task of this routine is to determine the new internal position based on the position where the previous primitive was matched. The routine will be presented for the OS mechanism.

In general terms, the adjustment of the internal position takes place in three phases. First the appropriate buffer must be determined, then the position in this buffer which is synchronized with the old one in the old buffer must be determined, and finally the new position must be determined by shifting one position to the left or to the right. In the following algorithm these three phases can be distinguished, although they are not fully separated:

```
function Select_int_pos(prim    : prim_type;
                        int_pos : internal_position)
                                : (internal_position, boolean);
begin {pre : lower_boundary < pos < upper_boundary}
   desbuf := Desired_buffer(patt_type, scan_dir, mod_type, prim);
   (buffer, pos) := int_pos;
   if (match_dir = ←)
   then pos := pos − 1;
        fail := (pos < lower_boundary) fi;
   if desbuf ≠ buffer and not fail
   then
      sync_list := Synchronization(int_pos);
      if (match_dir = →)
         then sync_val := Leftmost(sync_list)
         else sync_val := Rightmost(sync_list) fi;
      (pos, fail) := Search_sync(desbuf, sync_val)
   fi;
   if (match_dir = →) and not fail
   then pos := pos + 1;
        fail := (pos > upper_boundary) fi;
   int_pos := (desbuf, pos);
   Select_int_pos := (int_pos, fail)
end;
```

The first phase of determining the appropriate buffer depends on the pattern (focus, left or right context), the scanning direction (← or →), the type of

module (GTG, GTP, PTP) and the type of primitive (grapheme or phoneme), as described in section 4.4.1. Only the type of primitive can vary during the process of matching a pattern, and therefore only the type information is transferred to *Select_int_pos* by means of parameter *prim*. The other information is constant during the process of matching a pattern, and is therefore available inside the procedure by means of global variables, which are adjusted by higher level routines only when necessary. To emphasise the dependence they are included in the parameter list for the function *Desired_buffer*. This routine directly determines the appropriate buffer from these parameters, and is a straightforward implementation of the rules given in section 4.4.1.

If the thus determined buffer differs from the original buffer, the position in that buffer which is aligned with the original position in the original buffer must be determined by means of the synchronization information. The sync mark attached to a certain position is the sync mark behind the segment. If the matching direction is backwards (←), (which it is, for instance, when a left context is being matched,) the relevant sync marker is the one before the current position, and can be selected by first decrementing *pos*. The relevant sync marker can now be determined by selecting the first element of the sync marker list at that position (see 4.4.2). *Synchronization* selects the list, and *Leftmost* or *Rightmost* determines the first element, which again depends on the matching direction. Then, the sync mark is searched for in the new buffer by *Search_sync*. This routine searches for the value *sync_val* in buffer *desbuf* and returns the position where it is found, *pos*. Since nothing special is known of the sync marker, this is done by means of a linear search.

If the sync marker is not found, there is no synchronization between the two buffers at the original position. This means that the path of the pattern that is being matched is not present. This is transferred by the second value which *Search_sync* returns, *fail*, which is **true** if the search fails, and **false** if it succeeds.

Finally, the new internal position is determined. In the case of backwards matching (←) no further action is needed, due to the fact that the sync marks denote the synchronization points behind the segments. The decrementing action has been done beforehand, before the buffer switch has been made. In the case of forwards matching (→) the correct position is one position to the right, just across the synchronization mark, so in this case *pos* is incremented by one. As when decrementing, it should be checked whether the buffer boundaries have been crossed.

The variable *fail* thus performs a double task: it either indicates that there is no synchronization between the two buffers at the original internal position, or it indicates that the new internal position falls outside of the buffer range. In both cases the investigation of (that specific path of) a pattern can be

terminated, and since it is not important to know which of the two is the case, this can be coded by means of one variable.

4.4.5 Summary

In this section the implications of synchronized buffers for the matching algorithm *Match* have been discussed. First, the architecture of the synchronized buffers as is present in TooJiP has been shown and how this specific architecture affects the algorithm for matching primitives. Then a more general view on synchronization has been taken, that is, how an arbitrary number of buffers can be synchronized, what the characteristics of synchronization are, and how one can switch in general from one buffer to another. Next, two alternative mechanisms to implement synchronization have been presented, one prompted by the need for buffer switching, the other by the need for overall synchronization. They are functionally equivalent, but the main difference between the two is that the first mechanism is somewhat faster in run-time performance, whereas the second is a more simple and elegant algorithm as it directly codes the definition for buffer switching and overall synchronization. For this reason the latter mechanism is preferable, and is therefore implemented in the released versions of TooJiP. For this synchronization mechanism the algorithm which performs the task of buffer switching is given in *Sel_int_pos*, which is an important part of the routine for matching primitives.

4.5 Discussion

In the foregoing, the important aspects of how patterns can be built, what they mean, how they are represented internally, and how they are matched against synchronized buffers have been dealt with. Looking at the way patterns are processed, TooJiP can be viewed as a compiler/interpreter. The patterns, specified by the user in a high-level language, are compiled into their internal representations, which are then interpreted by *Match*. This is undoubtedly slower than full compilation of patterns into machine code, but it has an advantage that might be of importance to the future development of the system. The internal representations of the various patterns are stored in a dynamic structure in the order in which they are to be matched. Each pattern is a separate and traceable entry in that structure. This means that each such entry can in principle be replaced by a new one, for instance by a compiled version of an adjusted pattern. This could be done on-line, during a test session of the rules, which means that a linguist can interactively tune his rules. Such an extension of the system can be implemented relatively easily in the compiler/interpreter architecture of TooJiP, in contrast to a full compiler

system. Only the rule which undergoes a change has to be recompiled, which is typically so fast that it does not disturb the development session. The new internal representation then must be patched over the old one, which can be done instantaneously. Patching machine code, which would be the equivalent in the full compiler scheme, is generally viewed as a cumbersome enterprise, which requires detailed knowledge of the object code and is error-prone, and is therefore not to be recommended. Therefore, when slight adjustments must be made, generally in a full-compiler scheme the whole system is to be recompiled, which generally takes orders of magnitude longer than the incremental compilation in the compiler/interpreter scheme. Since TooIJP is first and foremost a development tool, such a feature of being able to test modifications quickly is interesting. The implementation of such a feature is planned in the near future.

On the other hand, the run-time performance of TooIJP is indeed that of an interpreter, that is, some 10 times slower than comparable fully compiled systems. It is difficult to give an exact figure of relative performance, since there is no system available with the same functionality that is fully compiled. The figure of 'some 10 times slower' is derived from a comparison with the SPE rule compiler of Kerkhoff, Wester & Boves (1984), called Fonpars, run on the same machine for the same input. In this comparison Fonpars differs from TooIJP in at least five respects. First of all, Fonpars is a compiler: the linguistic rules and modules are compiled into a Pascal program rather than some kind of dynamic internal representation. The Pascal program is then compiled into machine code in the usual way. The second difference is that Fonpars does not support complementation in the way TooIJP does. It does support a notion of complementation, but only for strings of the same length. This restricts the use of this operator, but on the other hand simplifies the algorithm which evaluates patterns. A third aspect is that Fonpars does not keep track of the derivation, and therefore does not make use of synchronization between buffers. A fourth point is that Fonpars does not include extensive interactive debugging facilities which are present in TooIJP (see 2.4.1 and 5.4.4). Finally, the rule sets which define the conversion differ in structure and magnitude. The Fonpars includes some 250 rules for grapheme-to-phoneme conversion which includes conversion of numbers, acronyms, etc., and the assignment of word accent. TooIJP, on the other hand, uses some 550 rules for the same purpose. The reason for this rather large difference in number has not been investigated. It is important to mention, however, that should the Fonpars rules be more concise, this is not an artefact of TooIJP. The formalisms supported by the two systems are comparable in power of expression (see 2.6.3). The relative contribution of each of these five factors is hard to measure exactly without considerable programming effort, and presumably not very interesting since these are only two of many systems

processing linguistic rules.

Comparison with existing similar systems is often a suitable manner to put into perspective one's own. As far as detailed implementation aspects are concerned, such as presented above, this is difficult to do, since in the literature there is no information available on this subject on other systems. Generally, only general characteristics of a system are described, such as how a user may formulate rules (the formalism), whether or not synchronization is used (not *how* this is implemented), whether it is a compiler-like structure or not, and so on. Such a functional comparison has been made in chapters 2 and 5.

4.6 Conclusion

In this chapter an algorithm is given which matches an arbitrary pattern against arbitrary input, where 'input' should be viewed in the general sense of synchronized buffers.

First, the internal representation is given into which the patterns, given by the user in this high level language, are transformed. Since the transformation process closely resembles the parsing phase of ordinary compilers, this has not been touched upon, so only the output of this process is given. The internal representation is more than a parse tree, but less than an NFA (non-deterministic finite automaton) with ϵ-transitions. This representation has been chosen on practical grounds; those computations which can be done at compile time are processed into the internal representation, those which are simpler during run-time are done at run-time.

Next, the interpreter for these (internally represented) patterns is presented, the function *Match*. This is done in two phases. First, the simplified case of a single input buffer, to which the pattern is to be matched, is discussed. Guided by the syntax and semantics of the patterns the various subroutines in the matching function are presented. Special attention has been given to the routine which deals with complementation. The recursion, which is directly present in the syntax, is interrupted at the highest level due to an efficient implementation of complementation. Once inside complementation, recursion is restored.

The second phase is the generalization to synchronized buffers. This only affects a small part of *Match*, the part which deals with the actual comparison of the buffers and the requirements imposed by the pattern. Two possible mechanisms to implement synchronization are presented and compared, one which is somewhat faster but more complicated, the other which is slower but simple and elegant. In a system which is in development and which for

instance could need to be extended in the respect of synchronizing capabilities, run-time performance is not a top priority, whereas transparency of algorithm is, and therefore the slower but elegant algorithm is preferable. For this mechanism the algorithm to switch from one buffer to another has been presented.

With respect to the processing of patterns TooJiP can be viewed as a compiler/interpreter. User-defined rules are compiled from high-level source code to an internal representation, which are further interpreted by *Match*. As a consequence, TooJiP is not as slow as an interpreter which interprets directly from source code, but not as fast as a compiler which compiles the source code directly into machine code. Nevertheless, an important characteristic of an interpreter has potentially been preserved, the possibility to modify a rule interactively and directly see the effect of the modification. This is an attractive feature of a development tool and is therefore planned to be implemented in the near future.

Appendix 4.A
Matching inside complementation

In this appendix the algorithms are given for matching structures inside a complemented structure. Of the four possible structures only for optionality has the algorithm been given in the main text (section 4.3.3). Therefore, the algorithms for alternation, simultaneity and complementation will be given here.

Alternation

```
function Exh_match_alt(patt      : pattern;
                       comm_list : result_list) : result_list;
begin
    int_pos := Int_pos(comm_list);
    prv_val := Res_value(comm_list);
    results := empty;
    alternative := Data(patt);
    while Present(alternative)
    do
        res_list := Exh_match(Pattern(alternative), int_pos);
        res_list := Adjust(prv_val, res_list);
        results := Unite(res_list, results);
        alternative := Select_next(alternative)
    od;
    Exh_match_alt := results;
end;
```

The structure of *Exh_match_alt* is similar to *Match_alt*. *comm_list* consists of only one element (see page 106) and serves to carry *int_pos* and *prv_val* into the routine. *prv_val* denotes the matching value of the path inside the complemented structure before the alternative structure was encountered. Compared to the non-complemented routine, the **while**-loop contains two extra statements. One is to include *prv_val* in the list that the current alternative has yielded. This is done by a routine called *Adjust*. This routine is not given, but consists of intersecting each individual result of the result list with *prv_val*. The second is the statement that prepares the result to be returned. This is done by the function *Unite*, which is given below.

```
function Unite(new : result_list;
               res : result_list) : result_list;
```

```
begin
    while Present(new)
    do
        ip := Int_pos(new);
        Select_first(res);
        while Present(res) cand ip ≠ Int_pos(res)
            do res := Select_next(res) od;
        if Present(res)   {This means : ip = Int_pos(res)}
            then res[result] := Res_value(res) or Res_value(new)
            else Append(new, res) fi;
        new := Select_next(new)
    od;
    Unite := res;
end;
```

The purpose of *Unite* is to combine the various result lists. If the internal positions of the various elements are not equal, they should both be included in the resulting return list. This is dealt with by the statement: *Append*(*new*, *res*). If the internal positions of two elements are the same they are united: *res*[*result*] := *Res_value*(*res*) **or** *Res_value*(*new*). The surrounding control structure serves to compare each element of the new list with the existing list.

Simultaneity

```
function Exh_match_sim(patt      : pattern;
                       comm_list : result_list) : result_list;
begin
    int_pos := Int_pos(comm_list);
    prv_val := Res_value(comm_list);
    results := empty;
    simultan := Data(patt);
    while Present(simultan)
    do
        res_list := Exh_match(Pattern(simultan), int_pos);
        res_list := Adjust(prv_val, res_list);
        results := Intersect(res_list, results);
        simultan := Select_next(simultan)
    od;
    Exh_match_sim := results;
end;
```

The structure of *Exh_match_sim* is symmetrical to *Exh_match_alt*. Instead of *Unite* the function *Intersect* is used.

```
function Intersect(new : result_list;
                   res : result_list) : result_list;
begin
   if not Present(res)
   then
      res := new  {Initialize}
   else
      while Present(new)
      do
         ip := Int_pos(new);
         Select_first(res);
         while Present(res) cand ip ≠ Int_pos(res)
            do res := Select_next(res) od;
         if Present(res)   {This means : ip = Int_pos(res)}
            then res[result] := Res_value(res) and Res_value(new)
            else res[new] := false;
                 Append(new, res) fi;
         new := Select_next(new)
      od
   fi;
   Intersect := res;
end;
```

Intersect is the counterpart of *Unite*. There are two significant differences. The main one is that each new internal position which is added to the resulting list is initialized **false**. Intersecting result lists means that internal positions must be equal and matching values must be **true**. If these are not both true, this may not lead to an explicit nofit. For the matching values this is accounted for by the statement: *res*[*result*] := *Res_value*(*res*) **and** *Res_value*(*new*). If the internal positions are not equal this means that the new internal position must be added to the list with a result value which is **false**. As a consequence, *res* must be initialized the first time *Intersect* is called. In *Unite* no explicit test is needed, since *Unite* simply adds new internal positions to the list without changing the matching values.

Complementation

```
function Exh_match_cmp(patt      : pattern;
                       comm_list : result_list) : result_list;
```

```
begin
    prv_val := Res_value(comm_list)
    cmp_list := Exh_match(Data(patt), Int_pos(comm_list));
    succ_patt := Select_next(patt);
    results := empty;
    while Present(cmp_list)
    do
        res_list := Exh_match(succ_patt, Int_pos(cmp_list));
        res_list := Cmp_adjust(prv_val, Res_value(cmp_list), res_list);
        results := Unite(res_list, results);
        Select_next(cmp_list)
    od;
    Exh_match_cmp := Cmp_simplify(results);
end;
```

Exh_match_cmp is set up in the same manner as *Exh_match_alt* and *Exh_match_sim*. There are three noticeable points. *Cmp_adjust* is similar to *Adjust* except that a facility has been included to keep track of the explicit nofits. In *Match_cmp* this was not necessary because of the sorted result list. Here, this is not possible since *all* the results must be returned instead of stopping when an explicit nofit has been detected. Therefore, the explicit nofits must be distinguished from regular non-matching paths. For this purpose **expnof** has been introduced as a third 'boolean' value. Only inside *Exh_match_cmp* the explicit nofits have to be known explicitly. Therefore, before returning the result list, the **expnof** values are simplified to **false**. As a consequence however, *Unite* must now be able to handle **expnof** values. Given their function this is defined as follows[2]:

```
expnof or true  = expnof
expnof or false = expnof
```

Cmp_simplify and *Cmp_adjust* are given below.

```
function Cmp_simplify(res : result_list) : result_list;
begin
    while Present(res)
    do if Res_value(res) = expnof then res.result := false fi;
        Select_next(res) od;
    Cmp_simplify := res;
end;
```

[2]The behaviour of **expnof** to '**and**' is not necessary here, but can be defined analogously.

```
function Cmp_adjust(first_part, com_part : boolean;
                    res_list             : result_list) : result_list;
begin
   while Present(res_list)
   do
      if first_part and com_part and Res_value(res_list)
         then res_list.result := expnof
         else res_list.result :=
            first_part and not com_part and Res_value(res_list) fi;
      res_list := Select_next(res_list)
   od;
   Cmp_adjust := res_list;
end;
```

Chapter 5

Evaluation

Abstract

In this chapter the merits of TooIJP are evaluated, and some recommendations for future development are made. Three sides of TooIJP are evaluated: (a) the outside, i.e., how TooIJP is used in a practical application, (b) the inside, i.e., how satisfactory was the semi-compositional formalism in practice, and (c) the surroundings, i.e., how does TooIJP relate to other systems which have been designed for similar purposes.

The main application for which TooIJP has been used is the design of a grapheme-to-phoneme conversion system. Two aspects of this application are discussed. The first is the spelling out of integer numbers, which is part of a pre-processing phase, the second concerns the linguistically slanted modules which perform the actual grapheme-to-phoneme conversion.

The second side of the evaluation concerns the use of the complementation operator. A semi-compositional formalism was devised to overcome the problems which occurred with respect to complementation in the compositional formalism. The usage of the complementation operator in the above practical application will be reviewed in the light of the choices which were made in chapter 3, and thus the validity of these choices are evaluated.

The third side concerns some more general aspects. For a number of features which can be seen to characterize development tools for linguistic rules, TooIJP is compared to seven important existing systems. This gives an overview of what is unique in TooIJP and what is common practice in such systems.

Finally, for the future development of TooIJP some recommendations are given which are concerned either with improving the system or extending it along natural lines.

5.1 Introduction

IN the previous three chapters a detailed inside view of TooIiP has been given. In chapters 2 and 3 the functional specification has been given and in chapter 4 the implementation. In this last chapter TooIiP will be viewed from somewhat more distance, once again, to put the system in some broader perspective. For three topics its merits will be investigated.

In the first place the major application for which TooIiP has been used will be discussed, viz. how it is applied as a grapheme-to-phoneme conversion system (section 5.2). Secondly, the complementation operator will be reviewed. Based on practical assumptions the semi-compositional formalism was implemented. The validity of these assumptions will be investigated, and with that the usefulness of the semi-compositional formalism (section 5.3). Finally, TooIiP will be evaluated as a development tool. The system will be reviewed in the light of some important features and is compared in those respects to some important similar systems (section 5.4).

By way of conclusion some aspects which are not yet included in TooIiP are discussed: some recommendations are made for extensions to TooIiP to increase its power and user-friendliness (section 5.5).

5.2 Applications

As reported in chapter 2, the main application for which TooIiP has been used is the development of a grapheme-to-phoneme conversion system for Dutch (Berendsen, Langeweg & Van Leeuwen, 1986). This application will now be discussed in greater detail than was provided in that chapter, without going into too much linguistic detail, however.

The conversion scheme is depicted in Fig. 5.1. It is subdivided into three major parts: text preprocessing, the actual grapheme-to-phoneme conversion, and some postprocessing. The text preprocessing serves to normalize the orthography, and concerns for instance the spelling out of numbers and acronyms. No full text-normalizing component has been achieved as yet, since this is quite an intricate problem. In principle, however, all input the normalizing component can handle is put out in a spelled out form.

This output is input to the grapheme-to-phoneme conversion, which consists broadly of three parts. First a morphosyntactic analysis takes place, which aims at locating the important morpheme boundaries. Next the spelling-to-sound conversion takes place, and finally word stress is determined. These processes are rule-driven. Exceptions are stored in a small exception

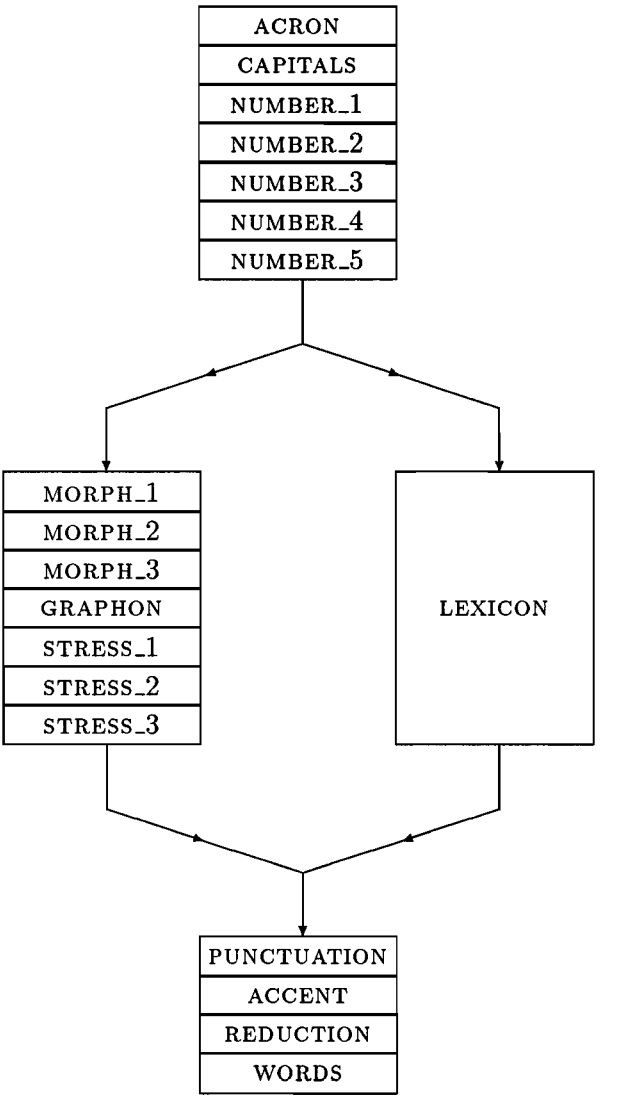


Figure 5.1: Modular composition of the main application, grapheme-to-phoneme conversion.

lexicon which is consulted first. The output of this grapheme-to-phoneme conversion is then sent to the post processing modules, the task of which is to deal with inter-word processes (such as assimilation) and to relocate sentence accent. Some of these tasks, such as assigning word stress or spelling out numbers, are divided into several physical modules. This is done because

some processes are 'crucially ordered', which will be elaborated on presently.

The less linguistically slanted modules, such as the spelling out of integer numbers and the relocation of sentence accent, have been devised by the author; the others have been devised by the users for whom the system is intended, linguists. The spelling out of numbers will be discussed in detail as an example of how specific tasks can be achieved with TooIJP. The linguistic modules will be discussed from somewhat more distance, for instance what type of constructs have been used frequently and which not. Finally some possible other applications are discussed.

5.2.1 Integer numbers

The rules

The spelling out of integer numbers is pre-eminently suited to be achieved by rule, since it is extremely regular. The automatic translation of numbers into Dutch has already been solved (Brandt Corstius, 1965), but the implementation into TooIJP was considered to be a good test case for the system.

To pronounce Dutch numbers correctly, two phenomena should be dealt with. The first is characteristic for arabic numbers, viz. the value of a digit depends on its position. This is not expressed explicitly in the orthography, '56', but in the pronunciation it is: 'fifty six' instead of 'five six'. The relative value of the digits will therefore have to be recovered. The second phenomenon is characteristic for Germanic languages, viz. the fact that the digits of numbers between 13 and 99 are pronounced in inverse order. Instead of 'fifty six' 'six and fifty' is said. So after determining the relative values, the 'tens' and the 'ones' should be inverted.

These two processes are the kernel of the number grammar. Since knowledge of the units is needed to invert some of the digits, this information must be available before inversion can take place. This motivates a separate module for each process. Beforehand some number normalization may take place, such as deletion of leading zeros and spurious dots (in Dutch '1,000,000' is written as '1.000.000'). Afterwards 'and' insertion (we say 'six and fifty' rather than 'six fifty') and the actual spelling must take place. Thus we have five modules, which will be discussed in order:

- NUMBER_1: for normalization
- NUMBER_2: for unit determination
- NUMBER_3: for digit inversion
- NUMBER_4: for 'and' insertion
- NUMBER_5: for spelling out

NUMBER_1: The normalization module currently consists of two simple rules. The first, (5.1), deletes all leading zeros in a number (**D** stands for a digit, # stands for a space). The second rule, (5.2), deletes all spurious dots. The module scans the input from left to right (→). Note that this scheme only functions satisfactorily for integer numbers, and that no account has been taken of other types of numbers, such as telephone numbers or numbers with decimals.

$$\textbf{zero} \rightarrow \textbf{0} \;/\; \# \_\_ \left\{ \begin{matrix} \textbf{D} \\ . \end{matrix} \right\} \tag{5.1}$$

$$. \rightarrow \textbf{0} \;/\; \_\_ \;\textbf{D} \tag{5.2}$$

A single '0' in the focus or change denotes an insertion or deletion. To refer to the single character '0' one must specify 'zero', as exemplified in (5.1). However, when the character '0' is part of a sequence there is no ambiguity and it can be used as any other character, as will be exemplified in rules (5.10) and (5.11).

NUMBER_2: As to the determination of units it can be observed that numbers naturally fall apart in groups of three (which for that reason are often separated by a comma in English and a dot in Dutch). The first group (seen from the right) are the ones, the second group the thousands, the third group the millions, and so on. Within each group of three, the leftmost digit denotes the hundreds, the middle digit the tens and the rightmost digit the ones.

The division into groups of three takes place from right to left, since the relative weight of a digit depends on the number of digits to its right. Therefore, assigning the relative weights to the digits is preferably also done from right to left.

For this purpose we define the module to scan backwards, i.e., from right to left (←). The input of the module will be a normalized number, for instance '1234567890'. The output must be a number where each digit is followed by an indication of its relative weight, thus the previous number must be put out as '1n2h3t4m5h6t7d8h9t0'. Here 'n' indicates milliard, 'm' million, 'd' thousand, 'h' hundred and 't' ten.

The rules included in Table 5.I perform the task. First some often occurring constructs are defined. **D** is an arbitrary digit, **B** is a boundary at the righthand side of a trio, and **DDD** is a trio into which the hundreds and tens are already inserted. Then the rules are stated. Each time, between two digits the appropriate unit is inserted. Tens will be inserted before any digit which has a trio boundary on its right side (the first rule). The first

time this will be a space, but the second time a 'd' (the thousands marker) will be present. Similarly, the hundreds are only inserted when the tens have just been inserted, the thousands only when a space (or other non-segment) delimits the trio, and so on. Note that this scheme makes use of information provided by a previously applied rule. Note also, because of the left contexts of all the rules, that only between two digits will unit-markers be inserted, and that only one rule at a time can operate.

NUMBER_3: With these unit-markers the task of inverting digits can be tackled. As stated, normally inversion takes place between 13 and 99, that is around any place where now a 't' is present. However, this is not the only place, for numbers consisting of 4 digits also exhibit this phenomenon. In English 'one thousand two hundred' is as correct as 'twelve hundred', but in Dutch the latter is preferable. This means that 1234, which is represented by 1d2h3t4 at this stage, should be transcribed to 2t1h4t3 in the next stage. Note that apart from inversion the units have changed as well. This string now codes twelve (2t1) hundred (h) four (4) and thirty (t3). Inversion around the thousands-marker 'd' also occurs for numbers bigger than one million, the digits of which for hundred-thousand and ten-thousand are zero: 1002300 is pronounced 'one million twenty three hundred'. If one of these digits is not zero, the thousands digit clings to the thousands trio: 1032300 is pronounced 'one million, two and thirty thousand, three hundred', and thus only the

Table 5.I: Module NUMBER_2: inserting unit markers.

definitions

$\#$ = <-segm>

D = <+dig>

$$B = \left\{ \begin{array}{c} \# \\ d \\ m \\ n \\ o \end{array} \right\}$$

DDD = D,h,D,t,D

insertions

0 → t / D _ D,B

0 → h / D _ D,t

0 → d / D _ DDD,#

0 → m / D _ DDD,d

0 → n / D _ DDD,m

0 → o / D _ DDD,n

'normal' inversion (within a trio) occurs. The normal and special inversion can be expressed by the following two rules:

$$\texttt{<+dig>}i\,\texttt{,t,<+dig>}j \;\rightarrow\; \texttt{<+dig>}j\,\texttt{,t,<+dig>}i \tag{5.3}$$

$$\texttt{<+dig>}i\,\texttt{,d,}\left[\begin{array}{c}\texttt{<+dig>}j\\ \neg\texttt{0}\end{array}\right] \;\rightarrow\; \texttt{<+dig>}j\,\texttt{,t,<+dig>}i \;/\; \left\{\begin{array}{c}\texttt{0,h,0,t}\\ \#\end{array}\right\} \_ \tag{5.4}$$

The first rule deals with the normal cases and inverts all digits around a 't'. The second rule deals with the thousands case and replaces the thousands marker by a tens marker. The exclusion of the zero in the focus ensures that 'multiples' of thousands are not included. The left context ensures that inversion only applies in the appropriate circumstances.

Since in TooIJP the i and j indices only select the unequal digits, one additional rule is needed for the 1100, 2200, etc. cases:

$$\texttt{<+dig>}i\,\texttt{,d,}\left[\begin{array}{c}\texttt{<+dig>}i\\ \neg\texttt{0}\end{array}\right] \;\rightarrow\; \texttt{<+dig>}i\,\texttt{,t,<+dig>}i \;/\; \left\{\begin{array}{c}\texttt{0,h,0,t}\\ \#\end{array}\right\} \_ \tag{5.5}$$

Note that for normal inversion such a rule is not necessary, since in the cases of 11, 22, etc., nothing needs to be changed.

This module, too, must scan from right to left (←). If it were scanned forwards, rule (5.3) would perform normal inversion inside the second trio of a number like 1002345 (.002... → .0h0t2d... → .0h2t0d...) which is not the intention, in this case. If the input string is scanned backwards, the desired effect will be achieved, since now rule (5.4) applies at the appropriate place and blocks rule (5.3) from working in the second trio. This is shown in the derivation below. The internal situation, when the inversion module is called for, is shown in (5.6).

input:	1 m 0 h 0 t 2 d 3 h 4 t 5	
	↑	(5.6)
output:		

Now, rule (5.3) applies, plus an additional copy action, which is performed since there are no rules which apply for the 'h':

$$\begin{array}{l|l|l|l|l|} \text{input:} & 1\ m\ 0\ h\ 0\ t\ 2\ d\ 3 & h & 4\ t\ 5 & \\ & \qquad\qquad\qquad\uparrow & & & \\ \text{output:} & & h & 5\ t\ 4 & \end{array} \tag{5.7}$$

Then, rule (5.4) applies:

$$\begin{array}{l|l|l|l|l|} \text{input:} & 1\ m\ 0\ h\ 0\ t & 2\ d\ 3 & h & 4\ t\ 5 & \\ & \qquad\qquad\uparrow & & & & \\ \text{output:} & & 3\ t\ 2 & h & 5\ t\ 4 & \end{array} \tag{5.8}$$

Rule (5.3) now cannot apply any more because the matching position has been advanced too much to the left. Thus, the other digits and unit markers will simply be copied.

NUMBER_4: The and-insertion is governed by the next module. There is one small point of attention. In Dutch we say 'six teen' to 16 but 'six and twenty' to 26. Only numbers above 20 have an 'and'-insertion. Further, multiples of 10 also lack the 'and'; we don't say 'zero and twenty', but just 'twenty'. Nevertheless, one rule suffices, (5.9):

$$\mathbf{t} \quad \rightarrow \quad \&\,,\mathbf{t} \quad / \quad \begin{bmatrix} \mathbf{D} \\ \neg\mathbf{0} \end{bmatrix} \quad \_ \quad \left[\neg \begin{Bmatrix} \overset{\mathbf{D}}{\mathbf{0}} \\ \mathbf{1} \end{Bmatrix} \right] \tag{5.9}$$

At this stage, the digits have been inverted, thus the restriction of a multiple of ten is imposed by the left context and the restriction of 20 and greater by the right context. The additional restriction that it concerns digits is to ensure that normal text containing a 't' will not receive an additional & (the symbol coding the 'and').

NUMBER_5: The module for spelling out the numbers has now become fairly straightforward. The units are marked, digits are inverted, the 'and' marker is inserted, and thus most of the symbols just code a sequence of letters. The only part which needs some attention is the units whose value is zero or one.

Generally, if a digit is zero, the accompanying unit marker should not be pronounced. This can be dealt with using the following rules:

$$\mathbf{0\,,h} \quad \rightarrow \quad \mathbf{0} \tag{5.10}$$

$$\mathbf{t\,,0} \quad \rightarrow \quad \mathbf{0} \tag{5.11}$$

$$\mathbf{zero} \quad \rightarrow \quad \mathbf{0} \tag{5.12}$$

When all three digits of a trio are zero, the unit marker accompanying the trio must not be pronounced either. This is accounted for by rule (5.13):

$$\mathbf{0,h,0,t,0,}\left\{\begin{array}{c}\mathbf{d}\\ \mathbf{m}\\ \mathbf{n}\\ \mathbf{o}\\ \vdots\end{array}\right\} \quad \rightarrow \quad \mathbf{0} \tag{5.13}$$

In contrast to all other zeros, a single zero must be pronounced ('nul' is the Dutch pronunciation for 'zero'):

$$\mathbf{zero} \quad \rightarrow \quad \mathbf{n,u,l} \;/\; \# \;\_\_\; \# \tag{5.14}$$

Finally, one extra rule is needed to deal with a special case. Numbers of the type 1002345 have a slightly deviating form when they enter this module: '1m0h0t3&t2h5&t7'. The '0' between the 'h' and the 't' denotes tens (of thousands). This is in contrast to all other digits before a 't', which denote the 'ones' due to the inversion. Only in this specific case, consisting of numbers with two zeros on the fifth and sixth digit (...D00DDDD), no inversion has been applied (in the trio of thousands) and therefore the zero *and* the 't' should be deleted:

$$\mathbf{0,t} \quad \rightarrow \quad \mathbf{0} \;/\; \quad \_\_ \quad \mathbf{D,(\&),t} \tag{5.15}$$

The right context specifies the circumstances in which this should happen. Only when a tens marker is followed by a digit and another tens marker is this the case.

When a digit's value is one, this number is generally pronounced. An exception to this is when it concerns hundreds and thousands. In Dutch we say 'hundred three and twenty' to '123' rather than 'one hundred...'. This is expressed in rule (5.10):

$$\mathbf{1} \quad \rightarrow \quad \mathbf{0} \;/\; \quad \_\_ \quad \left\{\begin{array}{c}\mathbf{h}\\ \mathbf{d}\end{array}\right\} \tag{5.16}$$

Here, too, one special case has to be dealt with. In numbers of the form '...D0010DD' the '1' may not be pronounced as it denotes 'one thousand' just like the previous case. The '1', however, has become separated from the 'd': '...Dm0h1t0d0hDtD', since normal inversion has been applied. Therefore, only if the '1' is enclosed by two zeros must it be deleted. However, by the time we reach the '1' in the rule base, part of the string will already be spelled out (the part to its left). In the left context we only have available the spelled-out form. Thus, the spelled-out form of 'h' (= honderd) may not be present,

since when it is, apparently the digit of the hundreds is not zero. In the right context the second digit, originally directly to its left, must also be zero. This is expressed in (5.17):

$$\mathbf{1} \rightarrow \mathbf{0} \;/\; \neg[\mathbf{h,o,n,d,e,r,d}] \_\_ \mathbf{t,0,d} \tag{5.17}$$

Now, spelling out the codes is trivial, and consists of some thirty rules of the following type:

$$\mathbf{5} \rightarrow \mathbf{v,i,j,f} \tag{5.18}$$

$$\mathbf{t,5} \rightarrow \mathbf{v,i,j,f,t,i,g} \tag{5.19}$$

The full rule set of module NUMBER_5 is included in the appendix.

Discussion

Looking at the functionality of the modules we can observe the following. The task of the first two modules, normalizing and determining the units, needs to be done in all languages using arabic numbers. Inversion of digits and 'and'-insertion, which is achieved by the next two modules, is necessary for Germanic languages. The last module is language-dependent but rewriting it for another language will mainly consist of translating the spelling of numbers. So, for instance, German numbers can be handled by translating the last module. The same goes for English numbers, only in this case the third and fourth module can be omitted.

It would be too strong a conclusion to put forward that spelling out numbers is solved for all languages using arabic numbers, since many languages will probably have their own specific exceptions. In French, for instance, numbers between 90 and 100 ('quatre vingt dix neuf') behave so differently from the numbers between 20 and 90 that this cannot be handled elegantly in the language dependent module. Probably this would motivate a separate '90-exception' module. The scheme, however, of describing different phenomena in separate modules is of course advantageous, and can probably be followed for a large number of languages.

As to the implementation in TooLiP some further observations can be made. In the first place the possibility of scanning the input from right to left is advantageous. In spite of the fact that we read from left to right in many languages, some phenomena can be dealt with better from right to left. Integer numbers, which have their anchor at the right side, are an example of this, but also stripping suffixes can be done best from right to left.

A second observation is that using the output buffer as a reference for those contexts for which it already contains information—as explained in the

previous chapter, and which for the user has the functionality of working with one string, in which the transformations are available directly—is generally advantageous, too. The unit insertion module advantageously makes use of this by referring to the last inserted unit marker. Thus the rules become simpler, and therefore more transparent and faster. Only in one case, rule (5.17), one might argue that the rule becomes more complicated than necessary.

A third observation is that metathesis, inversion of characters as exemplified in rules (5.3) and (5.4), is advantageous for a number grammar. In fact, it was first implemented when this application was developed, for in the conversion of normal words into a sound representation this had not yet appeared to be necessary (nor has it been used in the final set of rules). The alternative for rule (5.3) would in this case be an enumeration of 90 special case inversion rules of the type:

$$\mathbf{5,t,6} \quad \rightarrow \quad \mathbf{6,t,5} \tag{5.20}$$

which is, of course, not very elegant.

5.2.2 Linguistic modules

Functionality

The central part of the conversion system consists of the linguistic modules which perform the grapheme-to-phoneme conversion (see Fig. 5.1). This comprises a crude morphologic analysis, letter-to-sound transcription and stress assignment. The great majority of the 550 rules which define the total conversion are included in these seven modules.

The first three modules are a rule-based approach to the morphological problem. By means of rules it is attempted to determine the important morphological boundaries. No morpheme lexicon is utilized. Of course, in the morphological sense errors will be made, but for grapheme-to-phoneme conversion it does not seem necessary to have the full morphologic structure of the word available in all cases. Some boundaries are crucial in the sense that missing them will introduce a pronunciation error, other boundaries are not. The first three modules aim at locating these important boundaries.

Since prefixes and suffixes form a relatively small and closed group they can be identified reasonably elegantly by means of rules. Morpheme boundaries in compound words (which occur frequently in Dutch) can often be determined on morphosyntactic grounds. The sequence 'kp', for instance, does not occur in Dutch morphemes, and therefore a morpheme boundary between these two letters may be assumed. Combining the two, stripping affixes and applying

morphosyntactics, it appears (Berendsen, Lammens & Van Leeuwen, 1989) that only few crucial boundaries are missed.

The first 'morphological' module is the largest of all modules and scans the input from right to left (←). It strips the suffixes and inserts morpheme boundaries. The boundaries which are inserted before the suffixes, differ, where needed, in coding in order to indicate whether the suffix is stress-bearing, stress-attracting or stress-neutral. The second module scans its input (this is the output of MORPH_1) from left to right (→) and strips the prefixes. The last morphological module serves to delete some of the boundaries which were inserted wrongly by the previous two modules, which apparently were too difficult to exclude at the appropriate stage.

The fourth module, GRAPHON, deals with the second major task, the letter-to-sound transcription. Since most of the morpheme boundaries are placed correctly, this module consists to a considerable extent of relatively simple rules, without too many context restrictions. Quite some effort, however, is put into controlling the pronunciation of loan words, which in Dutch are extracted from several foreign languages.

The last task, assigning word stress, is separated into three modules, too. Rather than assigning stress to a syllable, stress is associated with vowels in this implementation, since accent-lending pitch movements are related to the vowel onset in Dutch ('t Hart & Cohen, 1973; 't Hart & Collier, 1975). Stress, in TooɪjP, is implemented by means of labels rather than by inserting special characters as is done with the morpheme boundaries.

The assignment of stress is more or less an implementation of the current theories on stress assignment in Dutch (Nunn, 1989; Kager, Visch & Zonneveld, 1987), expressed in the TooɪjP formalism. Not every notion of non-linear representation, such as binary feet, can be expressed in TooɪjP, but the effect of heavy and super-heavy syllables can be captured in the rules. In principle, primary and secondary stress are distinguished. In the first module, all vowels (except the schwa) are marked such that they can potentially bear stress. Then, in the second module, rather than directly marking primary stress, the appropriate vowels are marked with secondary stress, making use of the previously inserted morpheme boundaries. In the last module, the appropriate secondary stress labels are raised to primary stress and other vowels (which have not received any stress at all) are reduced or shortened.

Discussion

Since these modules constitute the main application, it is interesting to study the rules which constitute these modules, for they reveal which constructs are often used and therefore useful in practice, and which are not.

Two of the observations made for the rule grammar can also be made here. By having the applied transformations available directly, and by being able to scan the input in inverse order, suffixes can be stripped off recursively. The suffix boundary which is inserted by one rule triggers a next, similar rule.

Another observation is that macro patterns, which abbreviate frequently used constructs, are used extensively. Not only the more common **voc** (vowel) and **cons** (consonant) definitions, but also more complicated structures are used to indicate well-formed consonant clusters, syllable boundaries and vowel clusters.

Next, nearly all constructs which are available to the user are employed, the operators presented in chapter 2 are used frequently throughout all modules. In one of the stress modules, extensive use has been made of the possibilty to refer to graphemes and phonemes simultaneously. The two-dimensional notation in which alternation and simultaneity is expressed has been used to good advantage for devising patterns that would be utterly unreadable in a one-dimensional representation.

5.2.3 Other possible applications

Virtually any rule-governed segmental conversion scheme can be defined in TooLiP. Apart from the applications which have been discussed above, TooLiP can therefore also be used for other purposes. In the first place it can be used to describe or deal with other processes in a text-to-speech system, such as the recognition and spelling out of abbreviations, other types of numbers (telephone numbers, reals, etc.), or addresses. In the second place it can serve as part of other modules in the text-to-speech system. For instance, TooLiP could be used to perform morphological tasks, such as correcting for root mutation due to affixation, or deriving the singular forms from the plural. Also applications outside the text-to-speech range can be thought of. For example, one could build a module which can recognize syllables, with which one could build an automatic hyphenation machine. Another possibility might be to automatically determine inflectional forms. Or one can go the other way around and define phoneme-to-grapheme conversion. Also, a braille translation machine could be built, i.e., a program that translates text files into ASCII representative grade 2 braille files. Finally, also other rule-based symbolic transcriptions might possibly be expressed with TooLiP. For example, Bliss-to-text or Dominolex-to-text might be possible. Although these last applications are somewhat speculative, a Bliss-to-text system has been reported (Carlson, Granström & Hunnicutt, 1981), implemented with functions which are either present in TooLiP or can be simulated.

5.3 The complementation operator

In chapter 3 the introduction of the complementation operator in the semi-compositional formalism has been discussed. The semi-compositional formalism is a compromise between the practical needs of excluding the explicit nofits of a complemented structure and the theoretical elegancy of compositionality. It was decided to implement the semi-compositional formalism, since on the one hand it would 'succeed' in excluding the explicit nofits where the compositional formalism would 'fail', and on the other hand would fail only on patterns which are highly unlikely to be specified in practice. The patterns for which the compositional case would fail, however, are such that can be expected to be used in practice, and thus it was felt that TooIJiP should be able to handle these cases.

Now that the first major application has been completed, it is interesting to see how frequently the added functionality has been used. Added functionality means the functionality provided by the semi-compositional formalism which is not present in the compositional formalism. It is also interesting to see whether, despite their low probability, patterns are used for which the semi-compositional formalism fails.

It turns out that complementation is used quite frequently. In a total of some 700 rules the operator is used 214 times (for comparison: this is twice as often as the simultaneity, as often as optionality and one fifth as often as alternation). Its use can be subdivided into three classes. The first class is where complementation is used to exclude cases from a limited set, for instance as in pattern (5.21)[1]:

$$\left[\begin{array}{c}\mathbf{cons}\\ \neg\left\{\begin{array}{c}\mathbf{w}\\ \mathbf{h}\\ \mathbf{j}\end{array}\right\}\end{array}\right] \tag{5.21}$$

In other words, the universe to which the complementation 'operates' is provided by the user.

The second class is where complementation is used without explicitly specifying the universe and where the resulting pattern is consistent, as in pattern (5.22):

[1]These three examples, (5.21)–(5.23), are directly taken from the existing rulebase.

$$\left\{\begin{array}{c} \neg\mathbf{a}\,,\mathbf{a} \\ \neg\mathbf{o}\,,\mathbf{o} \\ \neg\mathbf{u}\,,\mathbf{u} \\ \neg[\mathbf{o}\,,\mathbf{e}]\,,\mathbf{i} \\ \neg\left\{\begin{array}{c}\mathbf{g}\\ \mathbf{b}\end{array}\right\},\mathbf{e} \end{array}\right\} \tag{5.22}$$

The third class is where complementation is used without explicitly specifying the universe and where the resulting pattern is inconsistent, such as pattern (5.23):

$$\neg\left\{\begin{array}{c}\mathbf{i}\,,\mathbf{n}\,,\mathbf{g}\\ \mathbf{voc}\,,\texttt{<-segm>}\end{array}\right\} \tag{5.23}$$

All complemented structures fall into one of these three classes. The first class comprises 50 % of all cases, the second 42.5 % and the third 7.5 %. This means that in 50 % of the cases the complemented patterns are naturally suited also for a compositional formalism where the universe is defined as $U = \sigma^*$. If the universe were indeed defined as such, complementation could only be used in practical situations if the universe is specified explicitly. Thus it turns out that in 50 % of the cases the user feels the need to do so anyhow. On the other hand it probably also means that in the other 50 % of the cases he is glad that he does not have to do so.

In 92.5 % of the cases the compositional formalism where the universe is defined as in Table 3.IX is satisfactory, since in the additional 42.5 % the universe is not specified explicitly, and only consistent patterns result.

Only in 7.5 % of the cases, 16 in total, does the functionality of the semi-compositional formalism appear to be necessary. This is not a large number, but nor is it negligible. No patterns have been specified of a complex form (for instance, double complementation of inconsistent patterns) for which the semi-compositional formalism is not adequate. This means that the practical choice of implementing the compromise between practical needs and compositionality has not been invalidated by this particular application.

5.3.1 Conclusion

On the whole it can be concluded that the choice of defining the universe as the character count of a pattern has been validated by practical use. Having to define the universe for each complemented pattern explicitly would needlessly burden the user and mistify the patterns. Furthermore, the choice to exclude explicit nofits by means of the semi-compositional formalism has on the one

hand not been invalidated, but on the other hand has not fully been validated either, since this functionality has been needed too infrequently. Given the theoretical objections to the semi-compositional formalism (not fully excluding explicit nofits, not being fully compositional, and its relative complexity), it can be argued that these objections and the increased computational complexity cannot be justified by the relatively small ergonomical advantages for the user. With some rewriting these few cases can probably be made suited for the compositional formalism.

5.4 TooIJiP in relation to comparable systems

In this section TooIJiP will be compared with a number of existing systems, most of which are used as the grapheme-to-phoneme conversion component of a text-to-speech system. The comparison will be guided by a number of properties which characterize the functioning and possibilities of such systems. First the systems will be introduced, and then each of them, including TooIJiP, will be viewed with regard to those properties.

The systems included in the comparison are all more or less high-level rule-based systems. Systems which approach grapheme-to-phoneme conversion, the main application of TooIJiP, from a different angle, such as a lexicon-based (Lammens, 1989) or morpheme decomposition approach (Pounder & Kommenda, 1986), differ too much from TooIJiP for there to be any point in comparing them. Also, other systems which are rule-based, but in which the rules are compiled by hand into a general-purpose programming language (Daelemans, 1987; Rühl, 1984) do not have enough interface with TooIJiP for an interesting comparison. Finally, some well-known text-to-speech systems, such as MITalk (Allen, Hunnicutt & Klatt, 1987), cannot be included in the comparison as the linguistic contents rather than the technique of implementation have been described in literature. In the following order, the systems to be compared with TooIJiP are: Rulsys, Fonpars, SPL, Depes, TWOL, Delta and Parspat. The first four systems, like TooIJiP, are based on SPE-based notations borrowed from generative phonology, whereas the latter three each have their own approach to rule-based conversion. The systems will now be introduced briefly.

Rulsys is the grapheme-to-phoneme conversion component of the multilingual text-to-speech system devised by Carlson & Granström (1976), (see also Carlson, Granström & Hunnicutt, 1989). It is one of the earlier systems which provide an environment for the development of linguistic rules. The rules are expressed by the user in a format borrowed from generative phonology, to be compiled and executed by the computer.

Fonpars is the system developed in Nijmegen by Kerkhoff, Wester & Boves (1984), especially for the linguistic front end of a text-to-speech system. Like Rulsys, the rules are expressed in a phonological format. Here, these rules are compiled into a Pascal program, which, in turn, needs to be compiled to obtain an executable program.

SPL was developed in Copenhagen by Holtse & Olsen (1985) and closely resembles the previous two. In correspondence with the above two systems, each rule operates on the output of the previous rule.

Depes was developed by Van Coile (1989) in Ghent and also belongs to the SPE-based rule compilers. Inspired by the Delta System, instead of having only one buffer available to store input, intermediate results and output, the system supports multiple layers in which related information can be stored and synchronized, for instance the relation between graphemes, phonemes and pitch movements. Also, control structures can be used to control the conversion process more freely.

TWOL was developed by Koskenniemi (1983) at the University of Helsinki (see also Karttunen, Koskenniemi & Kaplan, 1987) and originated as a morphological processor. It differs from the SPE-based systems in that the rules are of a declarative nature rather than imperative. The rules are therefore not ordered and can lead to conflicts; for instance, two rules which both apply may want to change a character differently. Such conflicts can be detected by a computer but must be resolved by the user.

Delta is the system developed by Hertz in Ithaca, N.Y. (Hertz, Kadin & Karplus, 1985). It was the first system to introduce a whole new approach to the linguistic front end of a text-to-speech systems. On the one hand the multi-layered, synchronized data structure, called a delta, was introduced, and on the other hand powerful control structures and functions to manipulate the delta became available. The customary notational conventions of generative phonology are lost, however, which is partly due to the multi-layered data structure.

Parspat is a system developed by Van der Steen (1987) in Amsterdam. Rather than devised only for grapheme-to-phoneme conversion the system is meant to generate programs for recognition, parsing and transduction. Grapheme-to-phoneme conversion is merely an application of this system. This system is interesting as it emerged from computer science rather than from the field of linguistics, and thus the solutions to some problems differ from those in other systems. Like TWOL, the transcription rules are declarative rather than imperative. An interesting aspect

of the system is that a complementation operator has been introduced fully in the regular expressions like formalism, be it in a different way than proposed in chapter 3.

These seven systems plus TooLiP will be discussed with regard to some characterizing properties, some of which will be subdivided. Not all systems can be viewed for all properties, since not every detail is reported for every system in the literature, but on the whole a reasonable overview can be given. Five main properties will be considered.

1. The first property concerns the formalism or language in which the linguistic knowledge must be expressed. This is subdivided into five characteristics: (a) whether the formalism is based on the formalism introduced by Chomsky & Halle (1968) in the Sound Pattern of English, which became very popular in generative phonology. This will be referred to as 'SPE-based'; (b) whether context specifications (patterns) are represented one-dimensionally or two-dimensionally; (c) whether and under what circumstances complementation can be used; (d) whether explicit control structures can be used to control the conversion process; and (e) whether numerical functions can be employed.

2. The second main property concerns the central data structure on which the formalism operates. This can be multi-layered or not. If it is multi-layered, the system provides information on different levels of representation, which are possibly co-ordinated. If the system is multi-layered, it will be discussed whether this induces the possibility of co-ordinated use of the different levels in the formalism, and whether it can be used to derive input-to-output correspondences.

3. The third property is the manner in which the linguistic rules are evaluated, i.e., if the rules are evaluated in order, if one can define the scanning direction, and what assignment strategy is applied.

4. The fourth property concerns whether or not a system provides support for the development of rules, and if so, which type of support is given.

5. The last property concerns some implementation features, viz. the computer language of implementation and the nature of the internal operator, i.e., whether it is a compiler or not.

5.4.1 The formalisms

As already mentioned, five of the systems are explicitly SPE-based, viz. Rulsys, Fonpars, SPL, Depes and TooJiP. TWOL has a representation which resembles SPE-type rules, but they are of a declarative nature. This means that they describe a relation between one representation and another rather than a description of how to derive the output from the input. In TWOL the application concerns the relation between a lexical representation (how the entries are stored in a morph lexicon) and a surface representation (how the entries are written, for instance wolf-s ↔ wolves). Delta and Parspat have more deviating rule formats. In Delta the rules are adapted to the programming language, which makes them more powerful but somewhat less readable. In Parspat the rules are expressed in the 'unifying formalism' described in Van der Steen (1987), which resembles the declarative BNF notation for Chomsky type-0 grammars. However, for both Delta and Parspat SPE-type rules can be transformed into the local formalism to produce the same effect. Except TooJiP, none of the systems feature a two-dimensional representation of patterns. As argued in chapter 2, a two-dimensional representation takes more space and (off-line) computational effort, but results in more transparent rules; vertical positioning denotes alternatives for the same position or co-ordination between two layers, and horizontal positioning denotes juxtaposition. For comparison, in each formalism the pronunciation rule for the 'c' which precedes an 'e' or an 'i' has been expressed in Table 5.II.

Most systems feature more or less the same operators to specify patterns as provided by TooJiP (see chapter 2). The complementation operator, however, is interesting since its introduction gives rise to unexpected problems, as discussed in chapter 3. Therefore, the presence and status of this operator has been investigated for the systems being compared. Not all reports in literature are as detailed, and in particular with respect to the complementation operation information is scarce, so the result will be presented with some reticence.

Most probably, SPL does not feature complementation as an operator. In an internal report (Holtse & Olsen, 1985) the language's syntax is described, which does not mention a complementation operator. Probably, the same goes for Rulsys, Depes and TWOL, but here the available information was less detailed, none of which contained any reference to complementation. In Fonpars, Delta and Parspat complementation is available. In Fonpars a restricted version is available in the sense that only constructions which contain strings of equal length may be complemented. This is more restricted than demanding patterns to be consistent, but syntactically less involved. In Delta, inconsistent patterns may be built, but from the documentation (Hertz, 1989) it is

Table 5.II: Comparison for expressing a particular rule in various formalisms.

Formalism	Expression	Remark
Paper & Pencil	c → s / — {e, i}	A 'c' is pronounced as an 's' if it is followed by an 'e' or an 'i'.
Rulsys	`crule c -> S / _ e` `crule c -> S / _ i`	`crule`: apply rule only once 2 rules: The presence of an alternative operator could not be established in the literature.
Fonpars	`c -> S / --- {e/i}`	`---`: focus mark `|`: separates alternatives
SPL	`replace c / _ e -> S` `replace c / _ i -> S`	`replace`: type indication of the rule. Structural change and context have been exchanged.
Depes	`|c| ---> |<2:s>| / _ {e,i}`	Focus and change field are enclosed by '|' markers. `c ---> <2:s>`: a 'c' in the default first layer (graphemes) will be associated with an 's' in the second layer (phonemes).

Formalism	Expression	Remark
TWOL	`c:s <= _ e: | i: ;`	A colon (:) separates the two layers. Therefore the rule states: a 'c' in the input layer is always (<=) associated with an 's' in the second layer, if the 'c' is followed by an 'e' or an 'i' in the input layer.
Delta	`_^1 c !^2 {e|i} ->` `  insert [s] ^1...^2`	_^1: the anchor of the expression, assumed to be positioned to the left of the 'c'. !^2: the sync mark to the right of 'c' is stored in ^2.
Parspat	`c,S,Rc1 :: c,Rc1.` `Rc1 :: e|i.`	BNF-like notation. The presence of the 'c' in the left-hand side of the first expression is not understood by the author.
TooljP	`c -> S / _ {e}` `{i}`	The 'or' braces have to be doubled to simulate the large enclosing braces.

unclear what the expressions mean. In Parspat complementation is defined exactly in accordance with Table 3.V, in other words in the compositional extension of regular expressions. As argued, this definition is not satisfactory for inconsistent patterns, but it is unclear whether this has been recognized in Parspat.

A fourth aspect is the possibility to use control structures, i.e., the possibility to explicitly control the conversion process. In generative phonology these are not present, and only in Depes and Delta are they fully available, due to the programming language character of the formalism. In Rulsys and SPL some control is possible by indicating whether the rule should be applied once or cyclically (Rulsys) or whether the rule is obligatory or optional (SPL), but this can hardly be counted as full control structures. In the other systems (except perhaps Parspat) flow of control is determined by the system itself and cannot be manipulated by the user.

A final aspect which characterizes the formalisms is whether or not numerical functions are available. For front-end linguistic processing this generally is not necessary, but in more phonetic processing, such as control of segmental duration, such functions are needed, and discrete segments are still the main units. Therefore, several systems which originated for linguistic purposes have evolved to be able to deal with more phonetically oriented rules. Examples are Rulsys, SPL and Fonpars. A future version of Delta has also been announced to support these facilities. The other systems either do not mention such facilities, or do not support them (TooɭjP).

5.4.2 Central data structure

A mutli-layered data structure is a convenient mechanism to be able to access derivational information in the rules and to be able to determine input-to-output relations. For this purpose the system should ensure that synchronization between buffers is reliable—this is the consistency requirement of 4.4.2. For instance, if a character is inserted in one layer, some of the characters in the other layer must be re-associated with the characters whose indices have changed.

Of the eight systems, Rulsys, Fonpars and SPL do not have a multi-layered data structure. TWOL and TooɭjP both have two layers and keep the synchronization consistent. Depes and Delta leave it to the user to define the number of layers. Parspat probably has a notion of multiple layers, that is, it could not be found explicitly in the literature, but in some examples it shows that one can refer to graphemes as well as phonemes within one rule. All systems with multiple layers support buffer switching. Depes and Parspat are unclear as to whether co-ordinated information can be used, such as referring

to an /o:/ which is derived from 'eau'; the other three support co-ordination. As to overall input-to-output relations, in Depes and Delta one can write a program to compute these. TooIJP provides an option to file these relationships. Parspat and TWOL are unclear as to this facility.

5.4.3 Inference mechanisms

Two different inference mechanisms are used to evaluate the rules in the systems which are included in the comparison. Declarative rules are evaluated in parallel, imperative rules sequentially. Declarative rules describe a relation between input and output, and can therefore often be used in two directions. In TWOL, for instance, the same morphological rules can be used to derive the surface structure from the lexicon structure (wolf-s → wolves), or to derive the possible set of lexical structures from the surface structure (wolves → wolf-s/wolv-s/wolves). Declarative rules are not ordered, that is, they are applied in parallel rather than sequentially. This has the disadvantage that rules can easily be conflicting and therefore have to be disjunct for the purpose of grapheme-to-phoneme conversion. For instance, the character 'c' can give rise to quite some pronunciations: /s/, /k/, /ʃ/, /χ/, etc. In sequential systems one can first deal with one case (for instance, when 'c' is followed by 'e' or 'i') and then consider the remainder of the cases—which means one does not have to consider the previous cases, since those have already been dealt with, and therefore the individual rules can be simpler.

On the other hand, imperative mechanisms only describe the path from input to output, not the other way around. Also they are much more deterministic by nature and therefore less suited to deal with ambiguities. Within the sequentially evaluated imperative systems there are two strategies. One is the 'rule-by-rule' mechanism, where the input string is completely dealt with by the first rule, before the output is dealt with by the next rule. The second mechanism is the 'segment-by-segment' mechanism where the first character consults all rules from top to bottom before the second character is dealt with. Both strategies can be useful, depending on the application. Finally, the possibility to determine the scanning direction is of interest, since for certain applications, such as stripping suffixes, scanning backwards is convenient.

TWOL and Parspat are of a declarative nature, evaluating the rules in parallel. The other systems are of an imperative nature, and hence the rules are inherently ordered. Most systems provide only rule-by-rule strategy, only Fonpars also provides a segment-by-segment strategy. TooIJP provides the segment-by-segment strategy, modules can be utilized for crucial ordering. Of the imperative systems most provide both forward and backward scanning, only SPL and Fonpars do not.

As to dealing with ambiguities, TWOL and Parspat are well suited to generate and handle them, due to their inference mechanism. SPL has a mechanism in the imperative setup to express the possibility of ambiguities. By means of defining a rule as 'optional', both the input and the output of the rule will be input for the following rules.

5.4.4 Development support

As the support which the various systems offer for rule development is scarcely documented in the literature, comparison of this characteristic cannot be complete.

Most systems, however, will probably feature some kind of syntax checking and accompanying error messages, since without it large sets of rules can hardly be debugged even on the syntax level. In TooLiP, when a syntax error has been encountered, the file, place and nature of the error are reported. In some cases, the error triggers some additional errors, but these will only occur within the rule in which the first error occurred. The other rules are parsed as if no error had occurred, so each first error of a rule can be taken seriously, and generally most error messages are of diagnostic value. In Delta, on the other hand, one syntax error generally triggers a whole lot of others (in the user manual the user is warned against this), so here generally only the first error message is diagnostic. This has the disadvantage that each real error will generally take an edit session and a compiler run before the next one can be identified.

Once a program (set of rules) is syntactically correct one is set to the task of making it semantically correct, that is getting the program to do what you want. Despite the rare occasions that this is reached at once, some effort is generally needed to reach it. The output of the program tells you something is wrong, but often does not give a clue about why and where it went wrong, so one will either have to reduce the program to the simplest version that still contains the error (in which case the error often is discovered) or one can advantageously use some debugging tools to analyse the larger and erroneous program directly. Only when the latter fails does one turn to the former strategy.

As to the debugging tools, two classes can be distinguished. One class is that of the tracing facilities, the other is often called the debugger. A tracer, when activated, reports the intermediate stages during conversion, reports which are generated by the system rather than by the user. Debuggers typically interrupt the program, whereupon the user can investigate the internal status by examining variables. Tracers are typically used in special-purpose programs, where flow of control is more or less the same for all applications,

and the number of central data structures is limited, so that special display routines can be devised. Debuggers are typically used in general purpose programs, where both control flow and data structures are user-determined. Tracers can often display the relevant information quickly and transparently, whereas debuggers only interrupt the program, whereupon the user must search the relevant information interactively. On the other hand, tracers are pre-programmed; if for some reason there is no access to sub-parts of the program in which the error happens to be located, the user has no other option but to reduce the program. Debuggers do have access to all parts, so generally the user never has to decide upon the laborious reduction of his program. Thus, on the whole, debuggers are less user-friendly but more powerful.

Of the eight systems, Fonpars, Parspat and TWOL do not report any debugging facilities. Depes and Delta feature a debugger where the breakpoints can be set and examined at run-time, so no compilation is needed for each new examination. Rulsys, SPL and TooIJP feature a trace facility. The exact nature of the trace facilities in Rulsys and SPL are not reported, but in both cases it is claimed that quite a detailed view of the actions performed by the system may be obtained.

On the trace facilities of TooIJP chapter 2 was not very specific and therefore some additional information will be given here. TooIJP distinguishes normal input from commands. Commands start with a dot followed by some mnemonic code indicating the type of command; '`.h`', for instance, gives help on the mnemonic codes by producing a list of commands which are available to modify the system's settings.

Locating a semantic error is done in three steps. First the module in which the error occurs must be determined. For this purpose '`.m`' must be typed, whereupon the intermediate results of each module are printed. The module in which the error is located can be identified. Next, the rule which is erroneous can be identified by invoking the tracer with '`.t`'. The user is prompted for the erroneous module. For that module a shorthand status report of each rule is given. If a rule has been applied its ranking number and its contribution to the output buffer is printed. If a rule has not been applied the pattern which did not match is printed: '`f`' is the focus pattern failed, '`l`' for left context, '`r`' for right context.

A more detailed view of a rule's operation can be obtained by invoking the second level of the tracer. Once again giving the command '`.t`' will achieve this ('`.t1`' and '`.t2`' are the explicit commands; '`.t`' increments the tracing one step to a maximum of two). Now, for a specific character and for a sequence of rules (for which the user is prompted) a detailed report on the rules operation is given. This may lead to a respectable amount of information (all primitive matching values are reported, as well as all operator

constructions and their conclusions), but generally the selection of the rules to be reported can be so limited that the information flow is not overwhelming, and on the other hand the report is minute enough to pinpoint the error. In the experience of users this three-step strategy is an efficient way to locate a semantic or conceptual error.

The final step in program development is, when it is semantically correct, improving the efficiency of the program. Rulsys reports that statistics of rule productivity can be gathered, the other systems do not report on tools for this purpose. As touched upon in chapter 2, TooIJP features a rule coverage analysis, which reports on the frequency with which individual rules are consulted. Like the trace facility, it can be invoked by a single command '`.lr`' (log rule coverage). The frequency rule consultations are then filed for all input cumulatively. Thus the infrequently consulted rules can be identified and be rewritten and rearranged. With this tool the efficiency of the existing grapheme to phoneme conversion rules has been increased by some 20 %.

5.4.5 Implementation aspects

As to the implementation, not many details are provided, either. Often, only a schematic and short section is included on the implementation, which states the language of implementation and gives a crude sketch of the system's architecture. Not much more than a list of facts can be given here. The choice of programming language is often influenced by the local conditions and acquaintance of the programmer with programming languages. One consideration which is heard is that the program should be transportable, but in this sense the generally known programming languages do not differ greatly.

SPL and Rulsys do not specify in which programming language they are implemented. TWOL is implemented in Lisp, Delta in C, and the other four systems in Pascal. As for TooIJP, apart from acquaintance with the language, a consideration has been that list structures and recursion is needed in extent, which is supported by Pascal. On the other hand it has been suggested that the unifying capabilities of Prolog might be very well suited for pattern matching. Apart from some uncertainty as to the execution speed, there seems no real reason why it could not have been implemented in Prolog as well.

All systems have been implemented as a compiler, except TooIJP which can be described as a compiler/interpreter. This topic has been discussed in the previous chapter, section 4.5. An advantage of an interpreter-like scheme is that on-line editing of rules is easily implementable. The price to be paid is often a slower execution of run-time. For the development of rules the

Table 5.III: Comparison between the eight systems on the basis of the properties discussed in the text.

	Rulsys	Fonpars	SPL	Depes	TWOL	Delta	Parspat	TooLiP
SPE-based	yes	yes	yes	yes	partly	no	no	yes
Two-dimensional	no	no	no	no	no	no	no	yes
Complementation	no	restr.	no	no	no	special	stfw.	full
Control structures	rudim.	no	no	yes	no	yes	no	no
Numerical functions	yes	yes	yes	no	no	future	no	no
Multi-layered	no	no	no	yes	yes	yes	prob.	yes
Co-ordination	-	-	-	?	yes	yes	?	yes
Derivational history	-	-	-	progr.	?	progr.	?	yes
Rule ordering	yes	yes	yes	yes	no	yes	no	yes
Scanning direction	?	$\rightarrow$	$\rightarrow$	$\leftrightarrow$	parallel	$\leftrightarrow$	parallel	$\leftrightarrow$
Assignment strategy	rbr	rbr/sbs	rbr	rbr	-	rbr	-	sbs/mbm
Ambiguities	no	no	partly	progr.	yes	progr.	yes	no
Syntax diagnosis	?	no	?	?	fragile	yes	?	yes
Trace facility	yes	no	yes	no	?	no	?	yes
Debugger	no	no	no	yes	?	yes	?	no
Efficiency support	yes	?	?	?	?	?	?	yes
Implemented in	?	Pascal	?	Pascal	Lisp	C	Pascal	Pascal
Compiler/interpreter	c	c	c	c	c	c	c	c/i

restr. = restricted
stfw. = straightforward
prob. = probably
rudim. = rudimentary
progr. = programmable

rbr = rule-by-rule
sbs = segment-by-segment
mbm = module by module
? = could not be established from the literature

first argument is more important, for operation in an application such as a text-to-speech system the latter may be of more importance.

5.4.6 Conclusion

The most important characteristics have now been reviewed. For an overview, they are included in Table 5.III for all systems included in the comparison. By way of conclusion the merits of TooIjP can now be enumerated. TooIjP is a system which fits in the SPE tradition. The rules are expressed in a formalism which is familiar to linguists, the main users of the system. Two features of the formalism are characteristic for TooIjP and are not present in other systems. One is the two-dimensional representation of 'and' and 'or' structures, and the other is the definition of complementation, the 'not' structure. Either the problem discussed in chapter 3 has not been recognized or it has not been elaborated on, but the solutions encountered in the various other systems are of a different nature of the solution in TooIjP.

Another characteristic of TooIjP is the support of a two-level data representation, one for graphemes, the other for phonemes. With it, grapheme-to-phoneme relations are available, both for usage in the rules and for statistical purposes. In this respect two other systems might be compatible, be it that for statistical purposes the user must probably write his own program. Furthermore, TooIjP provides relatively extensive development tools. Apart from clear diagnostic messages on the syntactic level and a powerful trace facility for semantic debugging, TooIjP also features a rule coverage analyser to improve the efficiency of a rule set. Most of these features are rarely touched upon by reports on the other systems.

Finally, TooIjP is restricted to deterministic symbol manipulation. It does not feature control structures in its formalism, nor numerical functions to specify for instance durational rules on the segmental level. From the input exactly one output is generated and thus ambiguous pronunciations cannot be generated. Furthermore, TooIjP does not provide any tools to represent tree-like structures which are typically desired in syntactic analysis of sentences. TooIjP, therefore, is suited for front-end linguistic processing in a text-to-speech system, which does not need tools to perform a complicated structural analysis. So typically TooIjP is suited for the applications it has been used for, such as spelling out acronyms, numbers, abbreviations, inserting boundaries on morphosyntactic grounds, performing grapheme-to-phoneme conversion and assigning word stress, but it is less suited to perform full morphological or syntactical analysis in order to improve grapheme-to-phoneme conversion or insert phrase boundaries and determine sentence accent.

5.5 Possible Extensions

This last section will discuss some possible future extensions of TooLjP. It comprises five topics concerned either with improving the system or extending it along natural lines.

5.5.1 Rule-by-rule assignment

As mentioned in chapter 2 and in the previous section, processes can be crucially ordered, that is one process must be finished before the other can apply. For this purpose TooLjP features modules; the next module will apply only when the current has fully been applied. This is the only mechanism in TooLjP to express crucial ordering. Despite the limited number of modules needed in a full-grown application it might be attractive to implement a more direct mechanism to express crucial ordering. In the current implementation some modules consist of only one rule, for instance NUMBER_4, as this rule must be applied after NUMBER_3 and before NUMBER_5. Also, a user who is accustomed to specifying in a crucially ordered manner does not want to create a module for each rule—which is a separate file for each module in the current implementation. The user retains more overview when such crucially ordered rules can be included in one file.

A possible solution to this is to allow the user to indicate for each group of rules in a module how to process them, either in the rule-by-rule strategy or the segment-by-segment strategy. Internally, the crucially ordered rules can be interpreted as a separate module for each rule which gives them the desired functionality, but for external (debugging) purposes the rule must be accessed via the module in which they are included. This will probably require little effort for the functional implementation but somewhat more for the user interface.

5.5.2 Simultaneous operator

One of the functions of the simultaneous operator is to express co-ordination between graphemes and phonemes, and it has been used rather frequently for instance for the purpose of stress assignment. While the operator is satisfactory for co-ordination and exclusion (such as "any consonant but 'c'"), the operator is not satisfactory for expressing non-synchronization, such as "the vowel 'II' which is not derived from 'ie'". Currently, when a pattern like

```
[   II   ]                                    (5.24)
[ '[i,e] ]
```

is expressed, this is interpreted as "the vowel 'II' which is synchronized with a sequence of two characters, the sequence 'ie' excluded". If the 'II' happens to be derived from a 'y' or a single 'i' this pattern will not match.

One possibility is to express this pattern positively (and 'II' can only be derived from a limited number of characters), but in other situations this may lead to patterns of undesirable complexity, and moreover, in the line of complete availability of negation this does not seem elegant.

Another possibility is to allow the user to explicitly denote non-synchronization, for instance in the following manner:

```
+[ II]
-[i,e]
```
(5.25)

Here, the minus in front of the square brackets denotes the fact that the 'II' may be synchronized with anything but 'ie'.

The introduction of such a mechanism involves some additonal theoretical work. Currently, the simultaneous operator is defined symmetrically to the alternative operator, that is, analogously with appropriate exchange of disjunction and conjunction operators. This also implies that the pattern succeeding the simultaneous operator is put into braces. For the purpose of synchronization and non-synchronization this might not be appropriate. Perhaps an explicit distinction on the level of user programming between simultaneity used as an 'and'-operator and simultaneity used as a synchronization operator is desirable. Detachment of the succeeding pattern from the synchronization operator, however, will have influence on the solution for the introduction of complementation, the exact nature of which will have to be studied closely before a satisfactory solution is found.

5.5.3 Extension of layers

In chapter 4 the architecture of a two-layered central data structure has been discussed, including the mechanism used to synchronize them. This architecture has been devised with the main application in mind, grapheme-to-phoneme conversion. The first layer contains graphemes, the second phonemes. Both data-types are user-defined.

With this, the kernel of a multi-layered data structure has been achieved. Extra layers can be added without too much effort if the OS synchronization mechanism is used to synchronize them. However, if the DBS synchronization mechanism is used, this will involve considerable effort.

The extension of layers is not restricted to segments; there is no reason why larger units such as morphemes (prefix/root/suffix) or words (lexical/number/abbreviation) should not be introduced. However, such an extension will also affect the formalism, which is the user's only tool for accessing the data structure. Currently, graphemes and phonemes are disjunct, so reference to either does not need disambiguation. When extra layers are defined, one possibility is that the elements they hold are disjunct to all other layers; another possibility may be that a layer selection mechanism is included in the formalism. The first solution may be undesirable in some applications, the latter may obscure the rules to an undesirable extent. A practicable solution might be a compromise of only having to disambiguate in ambiguous cases, and encouraging the user to use as few as possible equal sequences in different layers.

5.5.4 On-line rule editing

One of the characteristics of TooIjP is that the patterns are interpreted at a certain level. As argued in chapter 4, this property may well be capitalized by implementing on-line editing and testing of rules. This can probably be done with little effort, since the infrastructure is present. The rule to be edited, inserted or deleted can be selected in the same way as the rules to be traced currently are. The source text is available, so this can be read into the buffer of an editor. When the rule has been adjusted the system can compile the rule into its internal data representation and patch it over the old one so that it can be tested.

Of course, now the internal data differ from the original source. So a new source must be produced. Several solutions for this are possible. A safe and disk-space-friendly manner seems to be to store each change into a file with an indication which rule in which module it concerns. This serves as a journal file for recovery. Then, when the session is finished, like an editor, the system may ask whether the changes must be stored. For each module in which a rule has changed the system can generate the new source by appending all individual source texts.

5.5.5 Compiler implementation

An opposite extension is also possible, not serving the development tool but serving the run-time performance. When a rule set has been developed and performs satisfactorily, there is no reason why all development support tools should still be available. Therefore a full-compiler version of TooIjP is attractive for implementation in the text-to-speech system. On the one hand

all development support tools can be omitted, but the major part of the increase in run-time performance may be expected of the implementation of a full compiler.

One possibility is to follow the scheme of Fonpars, to translate the linguistic rules into a Pascal program, which in turn is compiled into machine code. An informal test, where a pattern was translated by hand into a Pascal program, indicated an increase of speed by a factor of 6, which seems a reasonable indication of the possible gain in speed with this approach. Another possibility is to translate the patterns into finite state machines as is done in TWOL. However this involves some theoretical work, since it is not clear beforehand how complementation should be translated to a finite state machine. An estimate of the possible gain in speed is hard to give, since the theoretical work has not been done yet, but it seems reasonable to believe that, since the approach is more direct, the increase in speed will be greater than in the first approach. However, the amount of effort will presumably be greater, too.

Appendix 5.A
Module NUMBER_5

In this appendix the full rule set of module NUMBER_5 is given. It serves to spell out the number when all important processes have been executed, viz. unit insertion, inversion and 'and' insertion. The rules follow below.

definitions

# = <-segm>
D = <+cijf>

grapheme 0

0,h,0,t,0, {d, m, n, o} → 0

0,t → 0 / __ D,(&),t

0,h → 0

zero → n,u,l / # __ #

zero → 0

grapheme 1

1 → 0 / __ {h, d}

1 → 0 / ¬[h,o,n,d,e,r,d] __ t,0,d

1,t,1 → e,l,f

1 → e,e,n

grapheme 2

2,t,1 → t,w,a,a,l,f

2 → t,w,e,e

grapheme 3

3,t,1 → d,e,r,t,i,e,n

3 → d,r,i,e

grapheme 4

4,t,1 → v,e,e,r,t,i,e,n

4 → v,i,e,r

grapheme 5

5 → v,i,j,f

grapheme 6

6 → z,e,s

grapheme 7

7 → z,e,v,e,n

grapheme 8

8 → a,c,h,t

```
grapheme 9
   9  →  n,e,g,e,n
grapheme &
   &  →  e,n
grapheme t
   t,0  →  0
   t,1  →  t,i,e,n
   t,2  →  t,w,i,n,t,i,g
   t,3  →  d,e,r,t,i,g
   t,4  →  v,e,e,r,t,i,g
   t,5  →  v,i,j,f,t,i,g
   t,6  →  z,e,s,t,i,g
   t,7  →  z,e,v,e,n,t,i,g
   t,8  →  t,a,c,h,t,i,g
   t,9  →  n,e,g,e,n,t,i,g
grapheme h
   h  →  h,o,n,d,e,r,d  /  __ D
grapheme d
   d  →  d,u,i,z,e,n,d  /  __ D
grapheme m
   m  →  m,i,l,j,o,e,n  /  __ D
grapheme n
   n  →  m,i,l,j,a,r,d  /  __ D
grapheme o
   o  →  b,i,l,j,o,e,n  /  __ D
```

References

Aho, A.V., Sethi, R. & Ullman, J.D. (1986). *Compilers; Principles, Techniques and Tools.* Addison Wesley, Reading, Massachussetts.

Ainsworth, W.A. (1973). A system for converting English text into speech. *IEEE Transactions on Audio and Electroacoustics*, **AU-21**, 288–290.

Allen, J., Hunnicutt, S. & Klatt, D. (1987). *From Text to Speech: The MITalk system.* Cambridge University Press, Cambridge.

Berendsen, E., Langeweg, S. & Van Leeuwen, H.C. (1986). Computational Phonology: merged not mixed. *Proceedings International Conference on Computational Linguistics 86*, 612–614.

Berendsen, E. & Don, J. (1987). Morphology and stress in a rule-based grapheme-to-phoneme conversion system for Dutch. *Proceedings European Conference on Speech Technology 87*, **1**, 239–242.

Berendsen, E., Lammens, J.M.G. & Van Leeuwen, H.C. (1989). Van tekst naar foneemrepresentatie, *Informatie*, *To appear.*

Brandt Corstius, H. (1965). Automatic translation of numbers into Dutch, *Foundations of Language*, January 1965, 59–62.

Carlson, R. & Granström, B. (1976). A text-to-speech system based entirely on rules. *Proceedings of ICASSP 76*, 686–688.

Carlson, R., Granström, B. & Hunnicutt, S. (1981). Bliss communication with speech or text output. *Quarterly Progress Status Report*, **4**, Royal Institute Of Technology, Stockholm, Sweden, 29–38.

Carlson, R., Granström, B. & Hunnicutt, S. (1989). Multilingual text-to-speech development and applications. *Internal Report*, Royal Institute Of Technology, Stockholm, Sweden.

Chomsky, N. & Halle, M. (1968). *The Sound Pattern of English.* Harper & Row, New York.

Daelemans, W. (1987). An Object-Oriented Computer Model of Morphophonological Aspects of Dutch. *Ph.D. Dissertation*, Katholieke Universiteit Leuven.

Elovitz, H.S., Johnson, R., McHugh, A. & Shore, J.E. (1976). Letter-to-sound rules for automatic translation of English text to phonetics. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **ASSP-24**(6), 446–459.

't Hart, J. & Cohen, A. (1973). Intonation by rule: a perceptual quest. *Journal of Phonetics*, **1**, 309–327.

't Hart, J. & Collier, R. (1975). Integrating different levels of intonation analysis. *Journal of Phonetics*, **3**, 235–255.

Hertz, S.R. (1981). SRS text-to-phoneme rules: a three-level rule strategy. *Proceedings of ICASSP 81*, 102–105.

Hertz, S.R. (1982). From text to speech with SRS. *Journal of the Acoustical Society of America*, **72**(4), 1155–1170.

Hertz, S.R., Kadin, J. & Karplus, K. (1985). The Delta rule development system for speech synthesis from text. *Proceedings of the IEEE*, **73**(11), 1589–1601.

Hertz, S.R. (1989). The Delta System. *User Manual.*

Holtse, P. & Olsen, A. (1985). SPL: a speech synthesis programming language. *Annual Report Institute of Phonetics*, **19**, University of Copenhagen, 1–42.

Hopcroft, J.E. & Ullman, J.D. (1979). *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, Reading, Massachussetts.

Kager, R., Visch, E. & Zonneveld, W. (1987). Nederlandse woordklemtoon (Hoofdklemtoon, Bijklemtoon, Reductie en Voeten). *Glot*, **10.2**, 197–220.

Karttunen, L., Koskenniemi, K. & Kaplan, R.M. (1987). A compiler for Two-level Phonological Rules. *Internal Report*, Xerox Palo Alto Research Center, Stanford University.

Kerkhoff, J., Wester, J. & Boves, L. (1984). A compiler for implementing the linguistic phase of a text-to-speech conversion system. *Linguistics in the Netherlands*, 111–117.

Kommenda, M. (1985). GRAPHON - ein System zur Sprachsynthese bei Texteingabe. In: *Österreichische Artificial Intelligence-Tagung*, (H. Trost, J. Retti eds.), Springer, Berlin.

Koskenniemi, K. (1983). Two-level Morphology: A General Computational Model for Word-Form Recognition and Production. *Ph.D. Dissertation, Publication no. 11*, Department of General Linguistics, University of Helsinky.

Kucera, H. & Francis, W.N. (1967). Computational Analysis of Present Day American English, Brown University Press, Providence, RI.

Kulas, W. & Rühl, H.W. (1985). Syntex—unrestricted conversion of text-to-speech for German. *New Systems and Architectures for Automatic Speech Recognition and Synthesis*, 517–535.

Lammens, J.M.G. (1987). A Lexicon-based Grapheme-to-phoneme conversion system. *Proceedings European Conference on Speech Technology 87*, **1**, 281–284.

Lammens, J.M.G. (1989). From text to speech via the Lexicon. *Internal Report, no. 7*, SPIN, Utrecht.

Lawrence, S.G.C. & Kaye, G. (1986). Alignment of phonemes with their corresponding orthography. *Computer, Speech and Language* **1**, 153–165.

Naur, P. (1963). Report on the Algorithmic Language ALGOL 60. *Journal ACM*, **6**, no. 1, 1–17.

Nunn, A.M. (1989). Het optimaliseren van de regelset van GRAFON, *Internal Report, no. 697*, Institute for Perception Research, IPO, Eindhoven.

Pounder, A. & Kommenda, M. (1986). Morphological Analysis for a German Text-to-Speech System. *Proceedings International Conference on Computational Linguistics 86*, 263–268.

Rühl, H.W. (1984). Sprachsynthese nach Regeln für unbeschränkten deutschen Text. *Ph.D. Dissertation*, Ruhr-Universität Bochum.

Van Coile, B.M.J. (1989). The Depes development system for text-to-speech synthesis. *Proceedings of ICASSP 89*, 250–253.

Van der Steen, G.J. (1987). A program generator for recognition, parsing and transduction with syntactic patterns, *Ph.D. Dissertation*, Faculteit der Letteren, University of Amsterdam.

Van Leeuwen, H.C., Berendsen, E. & Langeweg, S. (1986). Linguistics as an input for a flexible grapheme-to-phoneme conversion system for Dutch, *Proceedings IEE International Conference on Speech Input/Output 86*, 200–205.

Van Leeuwen, H.C. (1987). Complementation introduced in linguistic rewrite rules, *Proceedings European Conference on Speech Technology 87*, **1**, 292–295.

Van Leeuwen, H.C. (1989). A development tool for linguistic rules. *Computer, Speech and Language*, **3**, 83–104.

Summary

FOR the purpose of automatically converting (printed) text into speech, among other things grapheme-to-phoneme conversion is required, i.e., the assignment of a pronunciation code to the orthography. Since many words in a language are regular and the number of words in a language principally is not finite, a rule-based approach to this matter seems appropriate for at least a major part of the task. Exceptions, then, can be stored in a small lexicon.

In this thesis a tool is described for the development of linguistic rules with which one typically can define the transitions which are needed to derive the pronunciation from the orthography. The development tool is called TooLjP, which stands for "Tool for Linguistic Processing". And as the name suggests, although grapheme-to-phoneme conversion as yet has been the only major application for which TooLjP has been used, TooLjP is certainly not restricted to this application only. Probably any rule-based segmental transcription process can be implemented in TooLjP.

A special characteristic of TooLjP is that input-to-output relations are preserved. This means that one can make use of derivational information in the linguistic rules, so for instance for stress assignment rules this can be used advantageously. On the other hand it means that the system can be used to gather statistics on input-to-output relations. Given the major application for which TooLjP has been used, the system can be used as an analysis tool for statistics on grapheme-to-phoneme relations.

In this thesis TooLjP is treated from several points of view. In chapter 2 a user's point of view is taken, and TooLjP is described as it presents itself to the user. First the basic configuration is discussed. Linguistic rules are the user's main tool to manipulate input characters. With them, one can select and transcribe characters dependent on the context. The possibilities for transcribing input characters and the facilities for defining contexts are then described. Such rules can be grouped into a module, which thus provides a mechanism to manipulate strings. Modules, in turn, can be concatenated to form a conversion scheme, which performs the desired task. The chapter

concludes with a discussion on some extensions which are included to increase its user-friendliness and applicability. Also, some characteristics of the system are discussed and compared to those of some other systems.

In chapter 3 a mathematician's point of view is taken. It concerns a specific aspect of TooLjP, which remained underexposed in chapter 2, i.e., the exact meaning of patterns (the mechanism to denote sets of strings). In TooLjP patterns are an extended form of regular expressions. The extension consists of the addition of two operators to the standard regular expressions for user convenience: complementation (the 'not') and simultaneity (the 'and'). The introduction of one operator, the complementation operator, specifically gives rise to an unexpected problem. If the complementation operator is introduced in a compositional manner, the formal interpretation of a certain class of patterns differs from what one would expect them to mean. To be precise: certain strings one would expect to be excluded, are not. This is considered to be an undesirable characteristic, as generally users simply start using a system rather than first studying its exact nature. Therefore, an alternative definition for complementation is proposed, which for the mentioned class of expressions behaves in accordance with expectation. The essential difference with the compositional formalism is that now the 'explicit nofits' are always excluded. As a consequence, however, strict compositionality is lost, which shows, for instance, in the fact that double complementation may not always be annihilated. From a theoretical point of view the proposed formalism is thus not completely satisfactory. It might be satisfactory, however, from a practical point of view. Those patterns for which it behaves unsatisfactorily are highly unlikely to be used in practice, and the proposed formalism can be seen as a practical compromise between the practical needs and theoretical elegance. On these practical grounds it has therefore been decided to implement the proposed (semi-compositional) formalism in TooLjP.

In chapter 4 a technical point of view is taken, and some aspects of TooLjP's implementation are discussed. To be precise: those aspects of the implementation are described which concern the process of matching patterns to the input. Here, 'input' should be understood in the general sense of synchronized buffers, i.e., buffers of which the segments are aligned such that derivational information is available. For this purpose, first the internal representation of patterns is discussed. The user-specified patterns are transformed into a dynamic data structure which is accessible for the matching routine. The dynamic structure codes the structure of the patterns, but some simple adjustments are made also, which facilitate pattern matching during run-time. Next, the algorithms which perform the pattern matching are presented. First the situation of a single input buffer is considered. In view of this input situation the functions for matching a particular structure in a pattern are given. Special attention is paid to the function for matching the complementation

operator, since its definition gives rise to some additional computational complexity. Then the more general situation of synchronized buffers is considered. The algorithm for matching primitives is somewhat altered in this situation. Since the synchronization mechanism is important for this routine, two possible synchronization mechanisms are discussed and compared. The more general one is chosen to be implemented and the buffer switching algorithm is given. On the whole, with respect to the processing of patterns, TooIJP can be viewed as a compiler/interpreter. The user-defined patterns are compiled from high-level source code to an internal representation. The internal representation is then interpreted by the functions for pattern matching.

In the final chapter, chapter 5, the merits of TooIJP are evaluated. Three sides of TooIJP are considered: (a) the outside, i.e., how TooIJP is used in a practical application, (b) the inside, i.e., how satisfactory is the semi-compositional formalism in practice, and (c) the surroundings, i.e., how does TooIJP relate to other systems which have been designed for similar purposes? The main application for which TooIJP has been used is the design of a grapheme-to-phoneme conversion system. Two aspects of this application are discussed. The first is the spelling out of integer numbers, which is part of a pre-processing phase, the second concerns the linguistically slanted modules which perform the actual grapheme-to-phoneme conversion. The second side of the evaluation concerns the use of the complementation operator. The usage of the operator in the above application is reviewed in the light of the choices which were made in chapter 3, and thus the validity of these choices are evaluated. The usage of the operator in the specific situation that the compositional formalism would fail whereas the semi-compositional one succeeds is relative scarce. In the light of theoretical objections the choice for the semi-compositional formalism does not seem fully justified in this particular application. The third side concerns some more general aspects. For a number of features which can be seen to characterize development tools for linguistic rules, TooIJP is compared to seven important existing systems. This gives an overview of what is unique in TooIJP and what is common practice in such systems. The thesis is concluded with some recommendations for the future development of TooIJP. These concern both the improvement of the system and its extension along natural lines.

Samenvatting

OM langs automatische weg van gedrukte tekst naar spraak te komen is het onder andere nodig om over grafeem-foneem conversie te beschikken, dit is het toekennen van een uitspraakrepresentatie aan de geschreven vorm. Omdat veel woorden in een taal min of meer regelmatig in hun uitspraak zijn en het aantal woorden in een taal principieel niet eindig is, lijkt een aanpak die gebaseerd is op regels niet ongeschikt om een groot gedeelte van de woorden in een taal aan te kunnen. Uitzonderingen kunnen dan vervolgens in een kleine uitzonderingenlijst opgenomen worden.

In dit proefschrift wordt een gereedschap beschreven, dat bedoeld is om taalkundige regels te ontwikkelen, waarmee men typisch het soort overgangen kan beregelen dat nodig is om van spelling naar uitspraak te komen. Het ontwikkelgereedschap heet 'TooɪjP', hetgeen staat voor "Tool for Linguistic Processing" (gereedschap voor taalkundige verwerking). En, zoals de naam al aangeeft, ondanks het feit dat grafeem-foneem omzetting tot nu toe de voornaamste toepassing is waarvoor TooɪjP gebruikt is, is het systeem beslist niet beperkt tot deze enkele toepassing. Waarschijnlijk kan elke segmentele omzetting die door regels beschreven kan worden in TooɪjP geïmplementeerd worden.

Een speciale eigenschap van TooɪjP is dat de individuele relaties tussen de invoer en de uitvoer behouden blijven. Dit betekent dat men in de taalkundige regels gebruik kan maken van de afleiding tot dan toe, wat bijvoorbeeld bij de toekenning van woordklemtoon erg handig kan zijn. Anderzijds betekent het ook dat het systeem geschikt is om statistische gegevens te verzamelen over de relatie tussen de invoer en de uitvoer. Gegeven de toepassing waar het systeem voor gebruikt is, kan TooɪjP dus dienen als analyse gereedschap ter bepaling van welke orthografie hoe vaak tot welke klanken aanleiding geeft.

In dit proefschrift wordt TooɪjP van een aantal zijden belicht. In hoofdstuk 2 wordt een gebruikersstandpunt ingenomen, en wordt TooɪjP beschreven zoals het zich aan de gebruiker presenteert. Allereerst wordt de basisconfiguratie geschetst. De taalkundige regels zijn het middel om karakters te

manipuleren; ze worden gebruikt om karakters te selecteren en te wijzigen afhankelijk van de kontekst. De mogelijkheden om ze te herschrijven en de kontekst te definiëren worden vervolgens beschreven. Dit soort regels kunnen in een module gegroepeerd worden, om zodoende een mechanisme te vormen dat woorden of zinnen aan kan. Vervolgens kunnen modules weer in volgorde in een conversie schema geplaatst worden, dat zodoende de gewenste taak uitvoert. Het hoofdstuk besluit met de beschrijving van enkele uitbreidingen op de basisconfiguratie, bedoeld om de gebruikersvriendelijkheid en de toepasbaarheid van het systeem te verhogen. Tevens worden een aantal eigenschappen van het systeem besproken en vergeleken met die van soortgelijke systemen.

In hoofdstuk 3 wordt een wiskundig standpunt ingenomen. Het betreft een onderdeel dat in hoofdstuk 2 onderbelicht is gebleven, namelijk de precieze betekenis van patronen (het mechanisme om verzamelingen strings aan te duiden). In TooIjP zijn patronen een uitgebreide versie van reguliere expressies. De uitbreiding bestaat erin dat twee operatoren aan het formalisme zijn toegevoegd terwille van het gebruikersgemak, te weten complementatie (de 'niet') en coördinatie (de 'en'). De introductie van complementatie geeft specifiek aanleiding tot problemen. Als de operator op een compositionele wijze aan het formalisme wordt toegevoegd, wijkt de formele betekenis van een bepaalde klasse van patronen af van wat je zou verwachten. Om precies te zijn: bepaalde strings waarvan je zou verwachten dat ze uitgesloten worden, worden dat niet. Dit wordt onwenselijk geacht, omdat gebruikers in het algemeen een systeem gewoon gaan gebruiken zonder een voorafgaande studie van z'n exacte werking, zodat ze dan voor onverwachte en wellicht onnodige problemen komen te staan. Daarom wordt er in dit hoofdstuk een alternatieve definitie van complementatie voorgesteld, die voor de bewuste klasse van patronen wel volgens verwachting reageert. Het essentiële verschil met het voorgaande formalisme is dat nu de 'explicit nofits', de strings waarvan je expliciet verwacht dat ze uitgesloten zullen worden, nu wel altijd uitgesloten worden. De konsekwentie hiervan is echter wel dat strikte compositionaliteit verloren gaat, wat bijvoorbeeld geïllustreerd wordt door het feit dat dubbele complementatie niet zomaar altijd geschrapt mag worden. Theoretisch gezien is het voorgestelde formalisme dus niet geheel bevredigend. Praktisch gezien zou het dit echter wel kunnen zijn. Het is uiterst onwaarschijnlijk dat de patronen waarvoor het voorgestelde formalisme niet bevredigend reageert gebruikt worden in de praktijk, en zodoende kan het formalisme beschouwd worden als een compromis tussen praktische wensen en theoretische elegantie. Op grond van deze praktische overwegingen is er besloten om het voorgestelde (semi-compositionele) formalisme in TooIjP te implementeren.

In hoofdstuk 4 wordt een technisch standpunt ingenomen en worden een aantal aspecten van de implementatie besproken. Om precies te zijn: die

aspecten worden beschouwd die te maken hebben met het evalueren van patronen tegen de invoer. Hier moet 'invoer' gezien worden in de algemene zin van gesynchroniseerde buffers, dit zijn buffers waarvan de segmenten zodanig zijn opgelijnd dat de afleidingsinformatie beschikbaar is. Allereerst wordt de interne representatie van patronen besproken. De door de gebruiker gespecificeerde patronen worden omgezet naar een dynamische datastructuur die toegankelijk is voor de evaluatieroutines. De dynamische datastructuur codeert de structuur van de patronen, waar enkele kleine aanpassingen aan gedaan zijn die het evalueren vergemakkelijken. Vervolgens worden de algoritmes gegeven die de patronen evalueren tegen de invoer. Eerst wordt de situatie beschouwd alsof er maar één enkele invoer buffer bestaat. Voor die situatie worden de functies voor het evalueren van een bepaalde structuur gegeven. De complementatie operator wordt speciaal belicht omdat deze aanleiding geeft tot extra computationele complexiteit. Tenslotte wordt de algemene situatie van gesynchroniseerde buffers beschouwd. Het algoritme om primitieven te evalueren verandert hierdoor enigzins, maar de algoritmes voor structuren niet. Omdat het synchronisatiemechanisme belangrijk is voor genoemde routine worden twee mogelijke mechanismes besproken en vergeleken. Er is besloten om het algemenere mechanisme in TooLjiP te implementeren. Tenslotte wordt het algoritme gegeven om van buffers te veranderen. Over het geheel gezien kan TooLjiP als een compiler/interpreter gezien worden ten aanzien van het evalueren van patronen. De door de gebruiker in een taalkundig formaat gespecificeerde patronen worden gecompileerd naar de interne representatie. Vervolgens wordt die representatie geïnterpreteerd door de evaluatie functies.

In het laatste hoofdstuk, hoofdstuk 5, wordt TooLjiP op zijn merites beschouwd. Drie zijden van TooLjiP worden belicht: (a) de buitenkant, d.w.z. hoe is TooLjiP gebruikt in een praktische toepassing, (b) de binnenkant, d.w.z. hoe voldeed het voorgestelde formalisme in de praktijk, en (c) de omgeving, d.w.z. hoe verhoudt TooLjiP zich tot andere systemen die voor soortgelijke doeleinden zijn ontworpen? De voornaamste toepassing waar TooLjiP voor gebruikt is, is het ontwerpen van een grafeem-foneem omzettingssysteem. Twee aspecten van deze applicatie worden nader beschouwd. Enerzijds is dat een systeem om gehele getallen uit te schrijven, wat deel is van de voorbewerkingsfase, en anderzijds zijn dat de taalkundig getinte modules die de echte letter naar klank omzetting uitvoeren. Het tweede aspect van de evaluatie is het gebruik van de complementatie operator. Het gebruik van deze operator in de genoemde toepassing wordt bekeken in het licht van de keuzes die in hoofdstuk 3 gemaakt zijn, en zodoende wordt de juistheid van die keuzes geëvalueerd. Het gebruik van de operator in de specifieke situatie dat het compositionele formalisme fout gaat maar het semi-compositionele formalisme goed is betrekkelijk gering. De keus voor het semi-compositionele formalisme lijkt hierdoor gezien de theoretische bezwaren in deze toepassing

niet geheel gerechtvaardigd. De derde zijde betreft wat algemenere aspecten. Voor een aantal eigenschappen die voor dit soort systemen van belang zijn, wordt TooJiP vergeleken met een aantal belangrijke vergelijkbare systemen. Zodoende ontstaat er een overzicht over wat uniek is in TooJiP en wat gebruikelijk is in dit soort systemen. Het proefschrift wordt afgesloten met enkele aanbevelingen voor verder onderzoek. Deze betreffen enerzijds mogelijke verbeteringen aan het systeem en anderzijds uitbreidingen langs een natuurlijk lijn.

Curriculum Vitae

1 februari 1959	Geboren te Castricum
aug 1971 – juni 1977	St. Stanislascollege Delft. Gymnasium β.
sept 1977 – nov 1984	Technische Hogeschool Delft, Elektrotechniek. Afstudeerrichting: Regeltechniek. Afstudeeronderwerp: Ontwerp en realisatie van een interactieve, robotonafhakelijke robotprogrammeertaal.
jan 1985 – heden	Wetenschappelijk medewerker in dienst van het Natuurkundig Laboratorium van Philips, gedetacheerd op het Instituut voor Perceptie Onderzoek (IPO).

Stellingen

behorende bij het proefschrift
TooJiP: A development tool for linguistic rules
van Hugo van Leeuwen

1. In veel bestaande tekst naar spraak systemen worden verschillende niveau's van informatie in elkaar geklapt tot één enkele informatiestroom. Het is raadzamer deze verschillende niveau's expliciet weer te geven door middel van een gelaagde, gesynchroniseerde structuur, zoals die voor twee lagen besproken is in sectie 4.4 van dit proefschrift.

2. Bij het luisteren naar spraak valt het doorgaans zeer snel op of een spreker spontaan spreekt of voorleest. Een belangrijke oorzaak hiervan is dat een spreker die voorleest begint te spreken voordat hij/zij de zin geheel gelezen en begrepen heeft, en daardoor accenten legt die in stijd kunnen zijn met de inhoud van de tekst.

3. Dank zij de zgn. PSOLA-techniek, het eerst gerapporteerd door Hamon *et al.*, die het mogelijk maakt om zonder veel verlies aan spraakkwaliteit toonhoogte en temporele structuur van spraak te manipuleren, zal de kwaliteit van kunstmatige spraak op korte termijn een belangrijke verbetering in kwaliteit ondergaan.

   Hamon, C., Moulines, E. & Charpentier. F. (1989): *A diphone synthesis system based on time-domain prosodic modifications on speech,* ICASSP, 238–241.

4. Zowel in de overheidssector als in het bedrijfsleven worden veelal de budgetten voor een afdeling of werkgroep voor het komende jaar vastgesteld op grond van budget en uitgaven van het huidige jaar volgens de formule:

   $$B_{i+1} = Min(U_i, B_i)$$

   waar B het budget is, U de uitgaven, *Min* een functie is die het minimum van twee getallen bepaalt, en de indices de jaren in kwestie aanduiden. In tegenstelling tot de bezuinigende bedoeling hiervan is dit een verkwistende maatregel.

5. Momenteel kan een verdachte bloedafname voor identificatiedoeleinden d.m.v. de zgn. DNA-print weigeren door zich te beroepen op de onschendbaarheid van de lichamelijke integriteit. Het afnemen van vingerafdrukken kan de verdachte niet weigeren. Gezien de zekerheid waarmee de DNA-print identificeert, dient deze identificatiemethode dezelfde status te krijgen als de vingerafdruk, d.w.z. een verdachte dient het recht ontnomen te worden dit te kunnen weigeren.

6. Het is algemeen bekend dat specialisten en arts-assistenten in ziekenhuizen regelmatig onverantwoord lange werktijden maken. Als gevolg hiervan worden soms ernstige maar onnodige fouten gemaakt. Als specialisten van de toekomst rust er op de arts-assistenten speciale verantwoordelijkheid om aan deze misstand iets te doen. Zij dienen zich te verenigen om zodoende verantwoorde werktijden af te dwingen.

7. Het file probleem zou voor een deel opgelost kunnen worden door het fietsenvervoer in de trein gratis te maken en speciale faciliteiten op NS-stations te scheppen zodat men snel vanuit de trein per fiets uit het station kan komen en omgekeerd.

8. Het is de vraag of het merken van eigendommen een afdoende bescherming tegen diefstal biedt; de praktijk leert dat ze doorgaans ongemerkt verdwijnen.