City Research Online

City, University of London Institutional Repository

Citation: Cabrera Molina, J.J. (1985). English into Braille translation using augmented transition networks. (Unpublished Doctoral thesis, The City University)

This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/33185/

Link to published version:

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

The City University

Department of Computer Science

English into Braille Translation Using Augmented Transition Networks

by

Jose de Jesus Cabrera Molina, M. Eng.

Thesis Submitted for the Degree of Doctor of Philosophy

to The City University, Northampton Square, London

September, 1985.

Contents

Appendices

ABSTRACT

English into Braille Grade II Translation
Using Augmented Transition Networks.

A program has been written which translates English into Braille grade II using a formal language approach. Braille is a system used by blind people and it is the counterpart of what inkprint is to sighted people. As such, it allows the representation of their language, and the provision of "printed text". There are two types of Braille: grade I, which is a direct one to one mapping of inkprint characters into Braille, and grade II, which involves the use of contractions and abbreviations of words, in order to reduce the bulk of the material translated. This is the one considered in this thesis.

In the last few years, a number of programs have been developed for Braille translation; however, due to the difficulties inherent in the translation process, the problem is not yet completely solved. Even the professional systems use human proofreaders to validate the automatic translation. This implementation for performing Braille translation considers the use of Augmented Transition Networks, which are a development of pushdown automata. Its main advantage is that it offers great computational power which permits the representation of translation rules and their exceptions to these in a direct way.

The translation program is written in PASCAL and uses a small amount of memory resources, providing portability among different mini and microprocessors. The translator performs at about 2700 words per minute on a PDP-11/70 under RSX/11M operating system with an accuracy of over 99%, at 620 words per minute on a Wicat microcomputer, or at 50 words per minute on an Apple. The translator is trainable, thus providing the possibility of easily including new translation rules and exceptions to the existing ones by just altering the dictionary tables in a data file. There is no need to alter and recompile the program to handle the changes.

Acknowledgements

First of all I would like to aknowledge greatly the help I received from the British Council, which not only financed this research, but aided me wherever possible during the period it covered.

I would also like to thank my supervisor, Dr. Geoff Dowling, to whom I am indebted, for his enthusiasm, guidance and constant encouragement throughout the period of my research. I value greatly his advice in the structuring and writing of this Thesis, and also his friendship, which I appreciate deeply.

My thanks also go to Ali Syed for his help in copying the translation program into different micros at The City University.

In relation to Braille translation, the subject of this Thesis, I greatly acknowledge the help of the staff of the Royal National Institute for the Blind (R.N.I.B.), mainly Phil Coleman during the early stages of this research, and more recently, John Kaplin and Liam Madden. They helped to implement the translation program in their computer and provided the labour to generate and proof-read the program output.

I would also like to thank my friends in Mexico, especially Mr. E. Salcedo at Aseguradora Mexicana, Mr. S. Funes, Dr. S. Fuentes-Maya and Mr. M. Medina at the C.P.N.H., and Mr. A. Frost at BANAMEX for their assistance throughout the research project.

I want to thank also my wife Laura, for her patience and encouragement during these years of frequent week end and late night work, and also for her great help in the preparation of the final draft of this Thesis.

Finally, my thanks to my parents and grandfather who always encouraged me to continue with this research to its end.

Declaration

I grant powers of discretion to the University Librarian to allow this Thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Chapter 1

INTRODUCTION

One of the handicaps resulting from blindness is the severely restricted access to printed information. Some of the attempts to provide this type of information to the blind has been through the use of talking books and through the production of Braille. To this respect, at the present time, significant advances are occurring in the use of computers to augment the accessibility of this type of information which is needed by the blind for both employment and leisure activities (McGillivray, 1984; Gill, 1979; Davidson, 1979; RNIB, 1979; Silverman, 1980). This thesis is directed towards the solution of some of the problems encountered in the provision of printed material for the blind. More specifically, the aim of this thesis is to provide a translation program for English into Braille grade II, using a formal language approach. This approach was followed because the possibility of using a natural language translation-oriented approach, suggests a promising performance for English into Braille grade II translation.

This translator has been written with the idea of portability. The program is designed to run on microprocessors. This implies that the resource requirement for the translation program should be kept low, while the efficiency and reliability achieved must be high, for the

system to be adopted and thus be useful.

The translation program is easy to use. There is no need for the operator that keys in the printed text, to know about Braille or about computers. Thus, the keying in of the printed text is completely straightforward and in fact, many books are already stored in electronic form (Gill, 1979; RNIB, 1979). The only things to be specified by the operator are when a new paragraph is required, when italics are to be used, and when a specific word is not to be translated to grade II Braille (This occurs with foreign words, for example). The rules for the translation and their exceptions are given by means of a dictionary, thus providing the translator with flexibility in the inclusion or deletion of rules and exceptions. Ultimately, all this aims at the increasing of the amount of Braille material available, both for the community and the individual, thus providing better opportunities for the blind.

Initially, in chapter 2, a brief description of English Braille is given, with a review of computerised Braille translation. Then, in chapter 3, the translation problem is considered within the theory of formal languages. Comments are made on the types of grammars and augmented transition networks (ATNs) are identified within these grammars. In chapter 4, the transition network for English into Braille grade II translation is presented. Here the ATN used for the translation is described in detail. Then, with the groundwork given, the translation process is presented in

chapter 5. This chapter discusses the data structures used throughout the translation process and describes thoroughly each part of the translator. The actual translation algorithm is also included. The chapter concludes with some examples of the translation process by following in detail the corresponding translation algorithm. Together, chapters 4 and 5 form the spinal cord of this research project.

Afterwards, in chapter 6, comments on the characteristics and performance evaluation of the translator are given. This includes reliability, speed and flexibility. Finally, chapter 7 presents some conclusions related to the research project, and some thoughts in relation to possibilities for future development are given.

Seven appendices are included. Appendix 1 presents the contractions and abbreviations of English Braille grade II. Appendix 2 includes the mapping used for the translation of English Braille characters to ASCII characters. Appendix 3 presents a discussion related to finite state machines and pushdown automata, which are the theoretical basis of augmented transition networks. Appendix 4 presents the dictionary defined for the ATN for English into Braille grade II translation. It also presents the dictionary of exceptions. Appendix 5 relates to speeding-up techniques used throughout the translation process, mostly used to avoid useless tree-following so that there is always a high probability of finding a match. Appendix 6 gives instructions on program execution and the program listing,

and appendix 7 includes the set of tests used for the validation of the English into Braille translation program. Finally the Bibliography referenced is presented.

Chapter 2

ENGLISH BRAILLE

2.1 Braille Description

Braille is a system of printing or writing for the blind, invented during the first half of the nineteenth century by the French educator Louis Braille.

It is the most widely used means for providing the blind with printed material including books of text, mathematics, music and personal notes and letters. It is based on the Braille Cell, which consists of six dot positions (figure 2.1) spaced 2.5 mm apart, with adjacent cells spaced 4 mm apart.

1..4
2..5
3..6

A Braille Cell

Figure 2.1

In the current Braille system, information is conveyed by raising certain dot positions, so that in principle, 64 different patterns are available.

These cells are packed at less than 2 cells per square centimetre, whereas a packing density of 22 characters per

square centimetre is achieved with normal ink-print text, for which a greater character set is available and for which different sizes of characters can be easily used. The ratio of these two figures in conjunction with the thick paper necessary for Braille text, explains the weight, volume and to some extent the cost of the library of a blind person.

2.2 English Braille Grades I and II.

There are two kinds of English Braille: Grade I uncontracted, and Grade II contracted. Appendix 1 contains a brief description of Grades I and II of English Braille.

Grade I Braille uses one cell pattern for each letter of the alphabet, with some of the spare patterns being used for simple punctuation signs and abbreviations of a few very common words. Thus, it is a trivial matter to automate the translation of English text to Grade I Braille.

Grade II Braille makes extensive use of contractions to reduce the length of the text, with a resulting reduction in the number of pages of Braille, and for skilled users, in the reading and writing time. Commonly used letter sequences are contracted to one or two cell patterns. However, the automatic translation of text to English Grade II Braille, rather than Grade I, presents great difficulties if perfection is desired. To appreciate the nature of the difficulties, some specific problems are worth

mentioning:

- Position within the word.

  Some contractions are acceptable only at the beginning of the word; others, only in the middle of a word; others, at any place except at the beginning of the word.

- Position within the line.

  Some contractions may be performed only if the word that follows fits in the same line. In other cases, when hyphenating words, the contractions performed may depend on the place where the word is to be hyphenated.

- Adjacency to punctuation marks.

  Some contractions or abbreviations are allowable only when they are not in direct contact with punctuation marks.

- Grammatical construction of the word.

  Some contractions can only be performed when they do not break the grammatical structure of the word.

- Pronunciation.

  In some cases the reason for contracting or not contracting a certain word or part of a word depends on how the word is pronounced.

- Contraction Priorities.

  In many cases a choice has to be made regarding the selection of one out of two conflicting contractions. There are specific rules for the choice of contraction, but in some cases the underlying reasoning is difficult to define.

- Meaning of the word within the sentence.

  In some instances the only way to decide whether a contraction should be performed or not depends exclusively on the meaning of the word within the sentence.

As it may be observed, these problems cause great difficulties for automatic translation, and could suggest the need for a change; however, there are some important aspects that hinder this change. In fact, since its origination, it is remarkable how little the fundamental information element has changed.

Changes in the system are extremely desirable mostly due to the great achievements in technological development, which could permit easier learning by blind readers and sighted transcribers. It would also allow an easier generation of perfect Braille in an automated way.

However, the main reason for not having performed these important changes to the system has always been grounded on the fact that this sort of change would probably find

justified unacceptance by most of the people knowing the current Braille system. They would not be eager to learn a new system. This would oblige the Braille printing houses to maintain for a long period of time both the old and the new system, and this certainly is not economically feasible.

In spite of the difficulties just presented, the savings achieved with Grade II Braille result in a reduction of almost 30% in the number of cells required and also in the reduction of the number of skilled Braillists needed to proof-read the translated text. This has encouraged several groups to program translators for English into grade II Braille.

2.3 Computerised Braille Translation.

Two types of computer programs have been developed: the dictionary-based translator, and the translator based on rules and exceptions and a rather small dictionary.

In the first type of translator a complete dictionary of the language is stored, including hyphenating rules for each of the words stored in it. This sort of translator is of no interest from the development point of view, since all that it needs is a great amount of resources available to be able store and access the huge dictionary that it requires.

The second type of translator is the one to which greater

interest is to be given. This sort of translator uses a relatively modest amount of resources and tries to translate by systematic application of ordered rules, exceptions, and exceptions to the exceptions.

Several attempts have been made to program an English into Braille grade II translator, both in medium-sized and small computers. In the present project we shall concentrate on the work done using microcomputers, since the aim of this project is to provide a microprocessor oriented, portable translator. Considerable work has been already done in this field (Gill, 1979; Davidson, 1979; Gerhart, Millen and Sullivan, 1971; Silverman, 1980; Haynes and Siems, 1979; RNIB, 1979; McGillivray, 1984) but never with a formal languages approach, a fact that suggests the possibility of achieving better results than with previous attempts. Some of the work done in this field is described here to show the existing approaches.

The translation program written by Dr. Gill is aimed at producing a compact, inexpensive system for the production of contracted Braille, in which the operator is not required to know Braille. The system is easy to use, and generates a good approximation to Grade II Braille.

The translation program is controlled by a contraction table which is of the form:

columns 1-9	text string
column 10	previous character type
column 11	current character type
column 12	number of input characters
column 13	number of output characters
columns 14-18	output string

character types are:

L	letter
S	space or punctuation
N	number

Thus, this approach takes into consideration the context in which the input text appears, but only to a limited extent. It is able to take decisions according to the previous and the current character types, and generates a good Grade II Braille approximation. It may encounter problems however, when the context for taking the decision to contract or not extends beyond one character.

The translation program written by I. Davidson is aimed at generating a Grade II Braille translator suitable for microprocessors and is reported to generate a good approximation to Grade II Braille.

The search process for abbreviations and contractions is as follows:

For words having more than 10 characters, the first ten characters are compared with a table of abbreviations, and

if a match is found, then this group of letters is extracted and output in Braille form. If no match is found, the process is repeated with the last ten characters. If still no match is found, the process is repeated with a string of nine characters at the front and back of the word, and so on until the string is reduced to two characters in length. If this is unsuccessful, the first and last letters of the word are removed and translated into Braille, and the process is repeated until all the characters have been translated.

Though this approach takes into consideration the whole word to be translated, it does not take into account its surroundings, which are definitely important to perform the translation.

Another of the existing translators, and perhaps one of the most important ones, is DOTSYS-III, written by J. Sullivan. It is the translator in use at the R.N.I.B., and is written in FORTRAN. It can produce Grade II Standard English Braille from input text. The program will also produce grade I Braille, if it is required at any stage of the translation. All of the standard rules of Braille are implemented, except for word splitting at the end of the lines. The input text is interspersed with format codes to indicate page layout. As the program is table driven, irregularities of Braille can be incorporated as extra entries. Similarly, any variations in formatting requirements can be implemented by devising new format codes which use new combinations of existing facilities. A

tree-search algorithm is used for accessing the tables, and this results in a considerable time saving over a sequential search. The program uses a transition table, a decision table, and condition tables, and their counterparts can be seen in the translator developed for this thesis. However, the approach is different, since DOTSYS-III does not make use of formal languages. The translation achieved is good, though perhaps its main problem lies in the difficulties it presents when there is the need to modify or include new rules. In DOTSYS-III the rules are coded within the program.

As was mentioned earlier, there are several other translators, and if more information is required, it is well worth reading both (R.N.I.B., 1979) and (McGillivray, 1984), where a considerable number of translators are referenced.

At this point it may be convenient to specify some of the desirable features to have in a Grade II Braille translator.

- Portability. The English into Braille Grade II translator should be able to run on different computers, mainly in cheap microcomputers, so that the system may be made more easily accesible to a greater number of people.
- Accuracy. The translator should provide a good aproximation to Grade II Braille. Reported accuracy for existing translators is within the range of 90-100%. This feature may be sacrificed for speed.

- Speed. This depends on two components: the hardware and systems software supplied by the manufacturer, and the algorithm for performing the translation. Reported rates for English into Braille grade II translation are between 50 and 500 words per minute in microcomputers and 2000-3000 words per minute in medium-sized machines. This feature, as mentioned above, is often traded for accuracy.

- Trainability. When a translator is to be used over a prolonged period, it is useful to be able to add (or remove) entries in the table of rules and the dictionaries of abbreviations, contractions and exceptions. This allows the user to be able to keep the translator up to date, and obviates him from the necessity of delving into the code and modifying it.

In Chapter 6 of this thesis the performance of the translator is evaluated and a comparison with existing translators is made. For the moment it suffices to say that the translator developed for this thesis satisfies all of the stated features.

The translator described in this thesis is built using Augmented Transition Networks (ATNs), which are a development of the pushdown automaton. The formal treatment of the translation problem is described in the following chapters.

Chapter 3

AUGMENTED TRANSITION NETWORKS

3.1 The Decision to Use Augmented Transition Networks

The aim of the proposed English into Braille Translator, as stated in the previous chapter, is to provide a good approach to Grade II Braille, considering the computational characteristics of a microprocessor. With this in mind, the approach for translation by systematic application of ordered rules and exceptions, with a relatively modest dictionary, necessarily must be taken.

This suggested the possibility of using a formal language approach. To this respect, initially the concepts of grammars and recognisers will be introduced, and afterwards the English into Braille Grade II translation problem will be discussed with the types of grammars and state machines available for performing the translation.

Grammars are probably the most important class of generators of languages. A grammar is a mathematical system for defining a language, as well as a device for giving sentences in the language a useful structure.

A grammar for a language L uses two finite disjoint sets of symbols. These are the set of non-terminal symbols, denoted

by N, and the set of terminal symbols, denoted by Σ. The set of terminal symbols is the alphabet over which the language is defined. Non-terminal symbols are used in the generation of words in the language. The heart of the language is the finite set P of formation rules called productions, which describe how the sentences of the language are to be generated.

Next, the formal definition of a grammar is given following the formalism of Aho and Ullman. (Aho and Ullman, 1972)

Definition:

A grammar is a 4-tuple $G = (N, \Sigma, P, S)$, where:

N is a finite set of non-terminal symbols (sometimes called variables or syntactic categories).

Σ is a finite set of terminal symbols, disjoint from N.

P is a finite subset of formation rules (productions) in which the first component is any string containing at least one non-terminal and the second component is any string.

S is a distinguished symbol in N called the start symbol.

Chomsky defined several types of grammars according to the

formats of their productions. They are:

- Right Linear
- Context Free
- Context Sensitive
- Unrestricted

A right linear grammar is a generative system for simple languages, such that the parse (or syntax analysis) and translation depend only on the input read from the input tape and is processed sequentially. An example of this type of grammar is the one that generates numbers from single digits. This example is given next.

Example 3.1

Let G = ({<digit>}, {0, 1, ..., 9},
{<digit> -> 0|1|...|9|}, <digit>)

where Z -> A|B|C|...|N is a notational shorthand to denote the N productions:

Z --> A

Z --> B

.

.

.

Z --> N

Here, ⟨digit⟩ is treated as a single, non-terminal symbol. The language L(G) generated by this grammar is the set of ten decimal digits. Notice that L(G) is a finite set.

A Context Free Grammar (CFG) is a generative system mostly used for the definition of programming languages. It is a superset of right linear grammars. As its name implies, the productions generated do not depend on the context or position of an input character within the input tape. Once something is identified as part of the grammar from the input tape, it may be translated directly. A simple example of this type of grammar is the one that generates arithmetic expressions, within a programming language. This example is given next.

Example 3.2

Let G = ({E, T, F}, {a, +, *, (,)}, P, E)

where P consists of the productions:

E --> E+T|T

T --> T*F|F

F --> (E)a

An example of a derivation in this grammar would be:

E --> E+T

--> T+T

--> F+T

--> a+T

--> a+T*F

--> a+F*F

--> a+a*F

--> a+a*a

The language L(G) is the set of arithmetic expressions that can be built up using the symbols a, +, *, (, and).

A Context Sensitive Grammar (CSG) depends on the context in which input arrives from the input tape. The same input character may be parsed and translated differently depending on the context in which the input appeared. A CSG is recursive, and this is the reason why not every CFG is a CSG, since CFGs admit empty productions and CSGs do not.

However, every CFG without empty productions is a subset of a CSG.

By admitting empty productions in a CSG, the expanded grammars would be capable of defining recursively enumerable sets, which correspond to unrestricted grammars.

A second common method for providing a finite specification for a language is to define a recogniser for it. There are three components of a recogniser: an input tape, a finite state control and an auxiliary memory.

The input tape can be considered as a linear sequence of tape squares, each tape square containing exactly one input symbol from a finite input alphabet. There is an input head, which can read one input square at a given instant of time.

The memory of a recogniser can be any type of data store. The behaviour of the auxiliary memory for a class of recognisers is characterised by two functions: a store function and a fetch function. Generally speaking, it is the type of memory which determines the name of a recogniser. For example, a recogniser having a pushdown list for a memory would be called a pushdown recogniser (or more usually, a pushdown automaton).

The heart of the recogniser is the finite state control, which can be considered as a program which dictates the

behaviour of the recogniser. The control may be represented as a finite set of states together with a mapping which describes how the states change in accordance with the current input symbol and the current information fetched from memory.

The behaviour of a recogniser can be conveniently described in terms of the configurations of the recogniser. A configuration is a picture of the recogniser describing:

- The state of the finite control.
- The contents of the input tape, together with the location of the input head.
- The contents of the memory.

The finite control of a recogniser can be deterministic or non-deterministic. If the control is non-deterministic then in each configuration there is a finite set of possible moves that the recogniser can make. The control is said to be deterministic, if in each configuration there is at most one possible move.

The initial configuration of a recogniser is one in which the finite control is in a specified initial state: the input head is scanning the leftmost symbol on the input tape, and the memory has a specified initial content. A

final configuration is one in which the finite control is in one of a specified set of final states and the input head has moved off the input tape. Often the memory must also satisfy certain conditions if the configuration is to be final.

The language defined by a recogniser is the set of input strings it accepts. Each of these input strings presented to the recogniser in an initial state, will cause it to reach a final configuration.

For each class of grammars in the Chomsky hierarchy there is a natural class of recognisers that defines the same class of languages. These recognisers are finite automata, pushdown automata, linear bounded automata and Turing Machines. Specifically, the following characterisation of the Chomsky languages exists:

- A language is right linear if and only if it is defined by a finite automaton.

- A language is context free if and only if it is defined by a pushdown automaton.

- A language is context sensitive if and only if it is defined by a linear bounded automaton.

- A language is recursively enumerable if and only if it is defined by a Turing Machine.

There are a number of grammatical models that have been recently introduced outside the Chomsky hierarchy. Some of the motivation for introducing new grammatical models is to find a generative device that can better represent the syntax and/or the semantics of languages. This is true for Augmented Transition Networks, which are a further development of pushdown automata, having the same memory characteristics, but having additionally the possibility of performing arbitrary tests and actions according to the incoming information from the input tape and the information available on top of the stack.

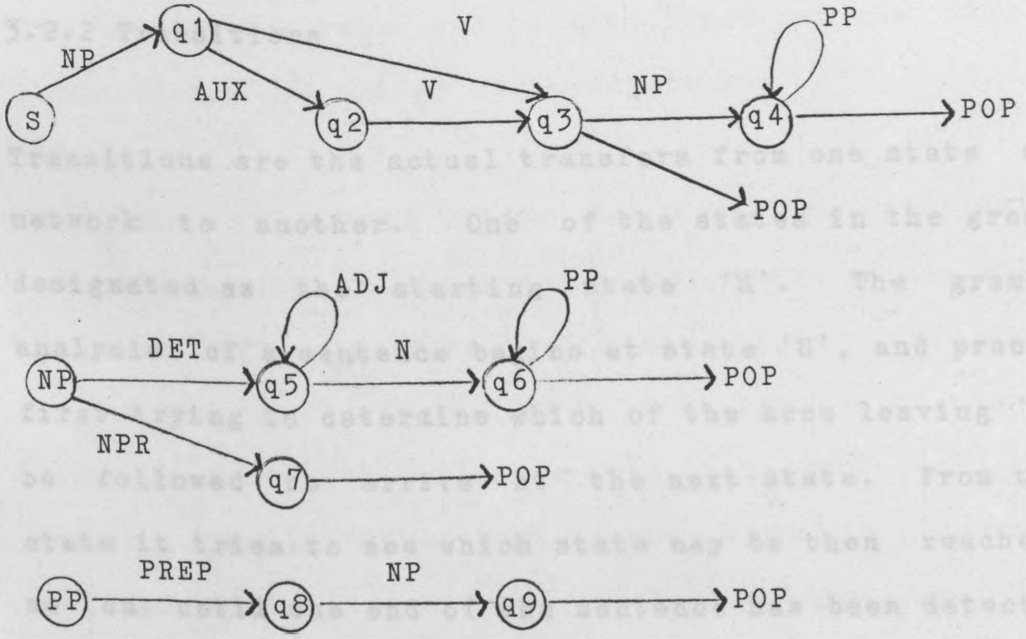
The reason for selecting this device is that the characteristics of the translation of English into Braille Grade II go beyond the scope of Context Free Languages, as there are productions that depend very much on the context in which the input appears.

Generally speaking, Augmented Transition Networks (ATNs) are meant to be used for natural language translation, (Woods, 1970) thus being able to deal with the subtleties involved therein. English into Braille Translation certainly is not as complex as natural language translation, but it is more complex than translating a programming language for which well-defined sets of rules exist. The next two sections will deal with a thorough description of ATNs; at the end of section 3.3 a set of examples of different uses of ATNs is given.

3.2 Theoretical Background for ATNs.

Transition networks are a useful way of representing linguistic structures. They permit a grammar to be presented through easily visualised diagrams and at the same time, they show the details of the restrictions and exceptions within the grammar.

There are some basic concepts related to transition networks: states, arcs, transitions, recursion and pushdown stacks. They will be illustrated by means of the simple transition network, shown in figure 3.1.



A Simple Transition Network

Figure 3.1

3.2.1 States and Arcs

The grammar given by the network in figure 3.1 consists of twelve states and eighteen arcs. The states are indicated by circles and the arcs by lines. Each state has its own label; arcs go from one state to another or from a state to the word 'POP', which will be explained later. Each arc is labeled either with the name of a word category like 'AUX' (for 'auxiliary') or with the label of another state of the grammar like 'NP', which begins the analysis of a construction within the grammar.

3.2.2 Transitions

Transitions are the actual transfers from one state of the network to another. One of the states in the grammar is designated as the starting state 'S'. The grammatical analysis of a sentence begins at state 'S', and proceeds by first trying to determine which of the arcs leaving 'S' can be followed to arrive at the next state. From the next state it tries to see which state may be then reached, and so on until the end of the sentence has been detected. If the end of the input sentence coincides with a 'POP' arc, the analysis is complete. Otherwise, the grammar is incomplete, or the input text was not a valid sentence and therefore is not accepted by the grammar. This will be illustated using the sample sentence: 'Mary likes the yellow dress'.

In the dictionary, 'Mary' is a proper noun (NPR), 'likes' is a verb (V), 'the' is a determiner (DET), 'yellow' is an adjective (ADJ), and 'dress' is a noun (N).

The arcs leading out of each state may be considered in any predefined order. This, in fact, allows fine tuning of the network, since the arcs that are likely to occur most often may be matched first.

The arc 'NP' is tried initially. As 'Mary' is a proper name (NPR), it forms a noun phrase, causing a transfer from 'NP' to 'q7' in the embedded 'NP' construct. As the 'NP' match was successful, the arc from 'S' to 'q1' is followed. The 'V' arc is then tried,and is also found to be successful, since 'likes' is not an auxiliary verb.

Next, the 'NP' arc is tried. The following input word is a determiner (the), which may start a noun phrase and so the embedded 'NP' network is followed. After the determiner comes to the word 'yellow', which belongs to the category 'ADJ', and as such, allows a transition from 'q5' back to 'q5'. Finally, the last input word is 'dress', which belongs to the 'N' category, initiating transfer from 'q5' to 'q6', and as the noun phrase is complete, a 'POP' is made to the immediate outer level, thus enabling the transition from 'q3' to 'q4'. As the end of the sentence has been reached, a final 'POP' is performed and the input text is accepted as a valid sentence according to the defined grammar.

3.2.3 Recursion

Recursion is a very important subject, and its application greatly increases the power of a grammar. Consider what would happen if instead of treating 'NP' as a construction by itself, there was an attempt to write it into the main network wherever it appeared. Notwithstanding the duplication that it would involve, it would not be possible, as the 'PP' construct contains an 'NP' within it, and 'NP' has an embedded 'PP' construct. It is at this point that recursion shows itself to be of great value, as it enables these cyclic constructs to be represented.

3.2.4 Pushdown Stacks

Whenever there is a construct to be recognised, rather than a simple word category that can be found in a dictionary, the transition network grammar saves the information obtained so far about the state of the network, on a special structure known as a pushdown stack. A new analysis is then started at the initial state of the subnetwork that represents the embedded construction. This is repeated every time there is an embedded construction. When a 'POP' arc is reached, the information found in the network in the current level is made available to the level immediately above. All the state information that had been found earlier for that higher level, is taken off the pushdown stack, and the process continues from that point.

This strategy for matching embedded constructs is called a 'PUSH' match, with every successful 'PUSH' leading to a corresponding 'POP'.

As shown in the previous example, the 'NP' construct was used for both the noun phrases 'Mary' and 'the yellow dress', without having to represent explicitly the 'NP' construct within the network.

It can be seen that there is only one accepting path through the network to the sentence 'Mary likes the yellow dress'. It is thus an unambiguous sentence within the grammar. It is an inherent feature of natural languages that there are ambiguous sentences that have more than one distinct analysis path through the transition network. A grammar that produces more than one parse tree for some sentence is said to be ambiguous, i.e., an ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for some sentence. For certain types of parsers, it is desirable that the grammar be made unambiguous; otherwise it cannot be uniquely determined which parse tree is to be selected for the input sentence. However, most of the times rules may be established which allow selection of only one parse tree, thus "disambiguating" the grammar. This will be illustrated next.

```
E --> E + E  |E - E|
     --> E * E  |E / E|  .
     --> E ^ E  |(E)   |
     --> - E
```

where E is a nonterminal abbreviation for 'expression'

This grammar is ambiguous. However, it can be easily "disambiguated" by specifying the associativity and precedence of the arithmetic operands.

Suppose it is desired to give the operators the precedences and associativities customarily used in programming languages[1].

Precedences in decreasing order:

```
-    (unary minus)
^
* /
+ -
```

---

[1]. These rules of precedence and associativity are common in most but not all programming languages.

Associativity:

^	right associative, i.e.,
	a^b^c implies a^(b^c).
* / + -	left associative, i.e.,
	a-b-c implies (a-b)-c

Therefore, with the aid of disambiguating rules, a transition network is a nondeterministic mechanism that must be capable of following any and all paths for a given sentence. Refer to Appendix 3 for more details on ambiguity (nondeterminism) of grammars.

Every recursive transition network is essentially a pushdown automaton whose stack vocabulary is a subset of its state set. Therefore, either a transition network or a pushdown automaton can represent successfully a context free grammar. However, it is well known (Woods, 1970; Aho and Ullman, 1972) that a strict context free grammar is not an adequate mechanism for characterising the subtleties of natural languages, and so an augmentation of this model is required to be able to deal with more complex structures.

3.3 Application of Augmented Transition Networks

The transition network model above is used as the basis for constructing Augmented Transition Networks (ATNs), by augmenting the original network so that the grammar is able

to handle agreement, order displacement and a more meaningful presentation of the results of the analysis. In fact, an ATN is a machine that is capable of doing everything that might be required of a grammar.

An ATN is a recursive transition network with three added features: registers, tests on arcs, and structure building actions that probably require several coupled pushdown stacks.

Registers may be likened to notes that are kept about items or constructions that match arcs. Whenever a 'PUSH' match is called for, all the registers that have been set up to that point in the network are saved on the pushdown stack, together with a record of the path that led up to the 'PUSH'. The analysis of the embedded construction then starts with empty registers and it fills up its own set of registers as it needs them. When the analysis of the embedded construction ends in a 'POP', the registers that were saved on the pushdown stack are popped, along with the associated path information, and the original network analysis resumes.

There is no restriction on how many registers may be set in going through a network, nor is there any predefined list of names appropriate for registers. They are created as needed by the linguist and labeled with one-word identifiers that serve as reminders for the kind of information they are storing.

Tests on arcs look for agreement. They do this most often by examining the contents of registers that have been set by earlier arcs. They may also be used to find out whether effort would be wasted by trying for a match that could be shown ahead of time to be bound for failure. The match specified for an arc will be tried only if the test on the arc is successful. If the arc does not pass the test, the match is not even attempted.

Actions are taken when a match is found. An arc can have any number of actions associated with it, to be performed if the match works, and ignored if it does not. The action most frequently taken upon finding a match is to set or modify the contents of one or more registers. Putting information into registers allows the transition network to reorganise the information later on, as the analysis proceeds. A transition network grammar can set up a tentative analysis, and change it later on, as new evidence comes in.

One class of frequent actions are the ones that permit a transition from the current state to another one, according to the tests performed. These are called terminal actions and can be found of two types; the first of these enables a transfer to another state ('TO' state), giving the name of the state to which a transition is to be made after the terminal action has been performed, providing the match is successful. This type of terminal action causes the following state to examine the next word of the input

sequence. The second type of terminal action, a 'JUMP' state, makes the named state further examine the current word, instead of simply moving ahead to process the next word in the input text.

Another common action is the one performed by 'POP' arcs, which may have fairly complex structure building actions associated with them. A 'POP' arc collects all the information about a construction that was put into the various registers since that part of the network was activated, and places it into a representation of the construction that has been found. In building this representation, the order of elements can be changed, and information can be either added or removed. The structure that is built up by the 'POP' arc is passed on to the arc that initiated the 'PUSH', which from then on, handles it as a unit.

Actions always follow tests on an arc, and end with a single terminal action. There is no limit on the number of actions associated with an arc.

Thus, the ATN formalism allows assignment to registers, arbitrarily complex tests on them, and the performance of equally arbitrary actions with the available information. These operations give an ATN its descriptive power and generality, and increase its computational power to that of a Turing Machine (Woods, 1979).

At this point it may be beneficial to comment on uses of ATNs. After Woods published his report on ATNs (Woods, 1969), he also developed a computer system that applied a transition network grammar to data in order to perform an automatic parsing or grammatical analysis (Grimes, 1974). The automatic parser was developed in connection with an information system for lunar geology. By feeding the analysis performed by the transition network grammar, into a semantic analyser, it was possible to have a computer find or calculate answers to questions about moon rocks that were asked in ordinary English rather than in an artificially constructed computer language. The grammar of English that was developed for this purpose, does not cover everything English speakers can say, but it does cover a surprisingly large proportion of the language using only 58 states and 172 arcs (Woods and Kaplan, 1972).

Another example of practical uses of ATNs is found in the work of N.G.P. Day (Day, 1979), who has used ATNs for building an editor. Specifically, this type of editor is one that advises the user of any departure from a given set of rules (defining, say, a programming language). It is based on a table-driven augmented transition network, and it may be used for example in checking the scope of variables when editing a language such as PASCAL.

G. Kaiser (Kaiser, 1981) has used ATNs to build an automatic extension of a knowledge base in the field of Artificial Intelligence. Specifically, a computer program

was written that acquires most of its knowledge from conversations among operators on Morse code radio networks. The system consisted of a learning component and a language understander. The learning component extended a 'core' augmented transition network knowledge base by generalising from sentences taken from scripts of actual conversations. The extensions enabled the understanding component to process a large number of sentences that are syntactically and semantically similar to the examples. The system's primary function was to parse and 'understand' human conversations in a simple language: it extracted the important information content of the conversations, updated particular items as necessary and 'forgot' any information that was made obsolete by information conveyed later in the conversation.

D. Bobrow and J. Fraser (Bobrow and Fraser, 1969) used ATNs for a syntactic analysis procedure, which obtained directly the deep structure information associated with an input sentence. The implementation used a state transition network characterising those linguistic facts representable in a context-free form, and a number of techniques to code and derive additional linguistic information and to permit the compression of the network size, thereby allowing a more efficient operation of the system. By recognising identical constituent predictions stemming from two different analysis paths, the system determined the structure of this constituent only once. When two alternative paths through the state transition network converged to a single state at

some point in the analysis, subsequent analyses were carried out only once despite the earlier ambiguity. Use of flags to carry feature concordance and previous context information allowed merging of a number of almost identical paths through the network.

Another example of use of ATNs is found in the field of data control (Linden, 1978). Linden's research was concerned with the development of a method of description and control of complex networks of data. The use of ATNs was proposed for this purpose; the main advantages of their use in this field were that they provided a consistent notation for the description of all constraints, and that they provided data independence in applications programs. This approach is valid in a wide range of application areas, and the formalism considered is independent of the underlying storage strategy.

Chapter 4

THE TRANSITION NETWORK FOR ENGLISH INTO BRAILLE GRADE II TRANSLATION

The transition network for English into Braille Grade II translation (EBT) is described next. The grammar definition is based on Wood's formal treatment of grammars (Woods, 1970).

Initially, the mathematical definition of the grammar will be presented; afterwards a graphic representation of the transition networks for the EBT is given, together with a description of the states therein comprised. Finally, a detailed explanation for all the transitions within the grammar is given, together with examples for the most important cases.

4.1 Mathematical Definition of the Grammar for English into Grade II Braille Translation

The Augmented Transition Network (ATN) defined for English into Grade II Braille translation will be established through the development of a pushdown automaton. This serves as the basis for the ATN, and the inclusion of a set of translation rules, which in practice constitute the arbitrary tests and actions, further develop this pushdown

automaton into an ATN.

Let $N = (Q, \Sigma, \Gamma, T, A, \delta, S, Z, F)$ where:

(1) $Q = \{S, C, D, E, \ldots, N\}$ Is the finite set of state symbols representing the possible states of the finite state control.

(2) $\Sigma = \Lambda \cup \Delta$ Is the finite input alphabet:

$\Lambda = \{A, \ldots, Z\}$

$\Delta = \{\text{delimiters}\}$

(3) $\Gamma = \{0, \ldots, 63\}$ Is a finite alphabet of pushdown list symbols.

(4) $T = \{t_1, t_2, \ldots, t_n\}$ Is the set of arbitrary tests that are required for a transition to exist.

(5) $A = \{a_1, a_2, \ldots, a_m\}$ Is the set of arbitrary actions to be taken for a transition to occur if the set of tests 'T' is satisfied.

(6) δ Is the mapping from $Q \times (\Sigma \cup \{e\}) \times \Gamma \times T$ to the finite subsets of $Q \times \Gamma^* \times A$.

e is the set that contains the

empty string.

Γ^* is the power set of Γ

(7) $S \in Q$ — Is the initial state of the finite control.

(8) $Z \in \Gamma$ — Is the symbol that appears initially on the pushdown list.

(9) $F = \{K, N\}$ — Is the set of final states.

One point that requires much further specification is (6) δ. The mapping from $Q \times (\Sigma \cup \{e\}) \times \Gamma \times T$ to $Q \times \Gamma^* \times A$ implies that it is possible to have for a specific case, any state (Q), any input character (Σ), including the null string ($\{e\}$), any pushdown list symbol on top of the stack (Γ), and a set of specific tests (T) to be satisfied for a transition to occur. This, in fact, includes the whole of the translation rules that must be followed to perform the actual mapping from English into Grade II Braille. It is at this point where the ATN proves useful, since it allows the specification of the translation rules, including the many different types of exceptions.

Each rule must be considered individually for each Braille abbreviation or contraction, and the suite of rules thus formed will represent the dictionary that must be searched in order to perform the translation.

At this point it is convenient to introduce some notation that will be useful for the establishment of the suite of transition rules for EBT.

A configuration of network N is a quadruple $(q, \omega, \alpha, \{t\})$ in $Q \times \Sigma^* \times \Gamma^* \times T$ where:

(1) q Represents the current state of the finite control.

(2) ω Represents the unused portion of the input. The first symbol of ω is under the input head of the automaton. If $\omega = \{e\}$, it is assumed that all of the input tape has been read.

(3) α Represents the contents of the pushdown list. The leftmost symbol of α is the topmost pushdown symbol. If $\alpha = \{e\}$it is assumed that the pushdown list is empty.

(4) $\{t\}$ Represents the specific set of tests that should be satisfied in order to enable a transition.

A move by N will be represented by $\vdash$ and is read "produces". One possible move by N is:

$$(q_i, b\omega, Z\alpha, \{t\}) \vdash (q_j, \omega, \gamma\alpha, \{a\})$$

assuming that $\delta(q_i, b, Z, \{t\})$ contains $(q_j, \gamma, \{a\})$ for $q_i, q_j \in Q$, $b \in \Sigma \cup \{e\}$, $w \in \Sigma^*$, $Z \in \Gamma$, $\alpha \in \Gamma^*$, $\{t\} \in T$.
This move means that if there is a mapping 'δ' that allows a transition from state 'q_i', given input 'b', topmost pushdown symbol 'Z', and tests 't', the transition will occur to state 'q_j'. The unused input 'w' remains unmodified, the topmost symbol 'Z' is changed to $\gamma \in \Gamma$, and the pushdown list 'α' usually remains unmodified, unless the arbitrary actions {a} A affect the stack contents. These concepts and this notation will be useful in the following sections, where the network is depicted and complete examples of the transitions are given.

4.2 Network Description

The following aspects are observed when applying ATNs to English into Braille Grade II translation.

1. The ATN is defined as a list of arc sets.

2. Each of these arc sets is a list whose first element is a state name and whose remaining elements are arcs leaving that state. Each arc set represents one step in the EBT process.

3. Associated with each arc there is a number of arbitrary tests that must be satisfied in order to follow the transition network. Also, there are

associated actions that indicate what to do with the input character and with the stacks' contents. These depend on the results of the tests performed.

The ATN for EBT is based on the following 11 simple subnetworks. The approach of representing the network separately rather than as a whole has been selected for ease of understanding. Also, in this way, the notion of recursion within the network is represented more clearly.

4.2.1 Transition Network for English into Braille Grade II Translation

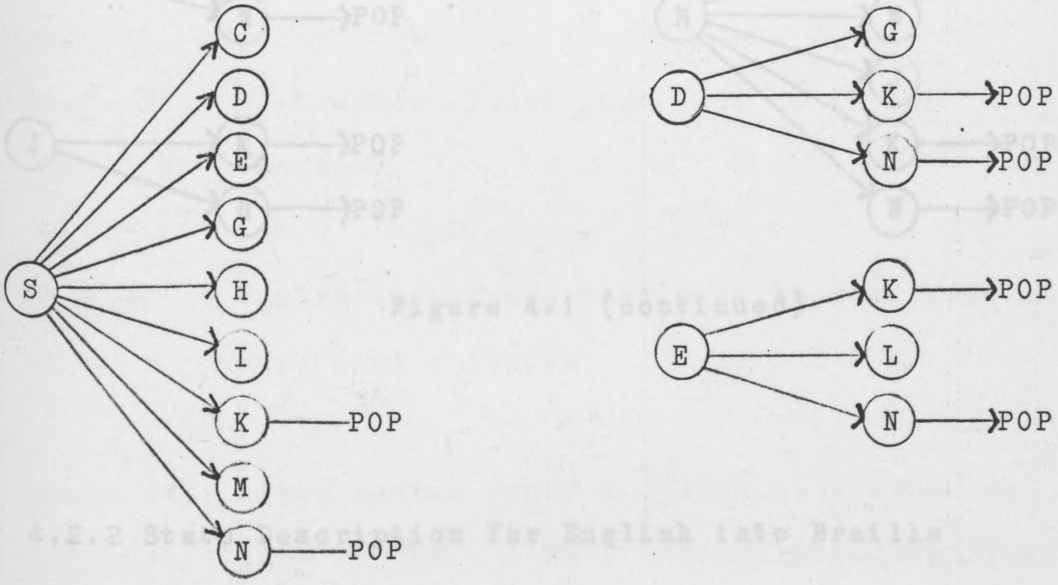


Figure 4.1

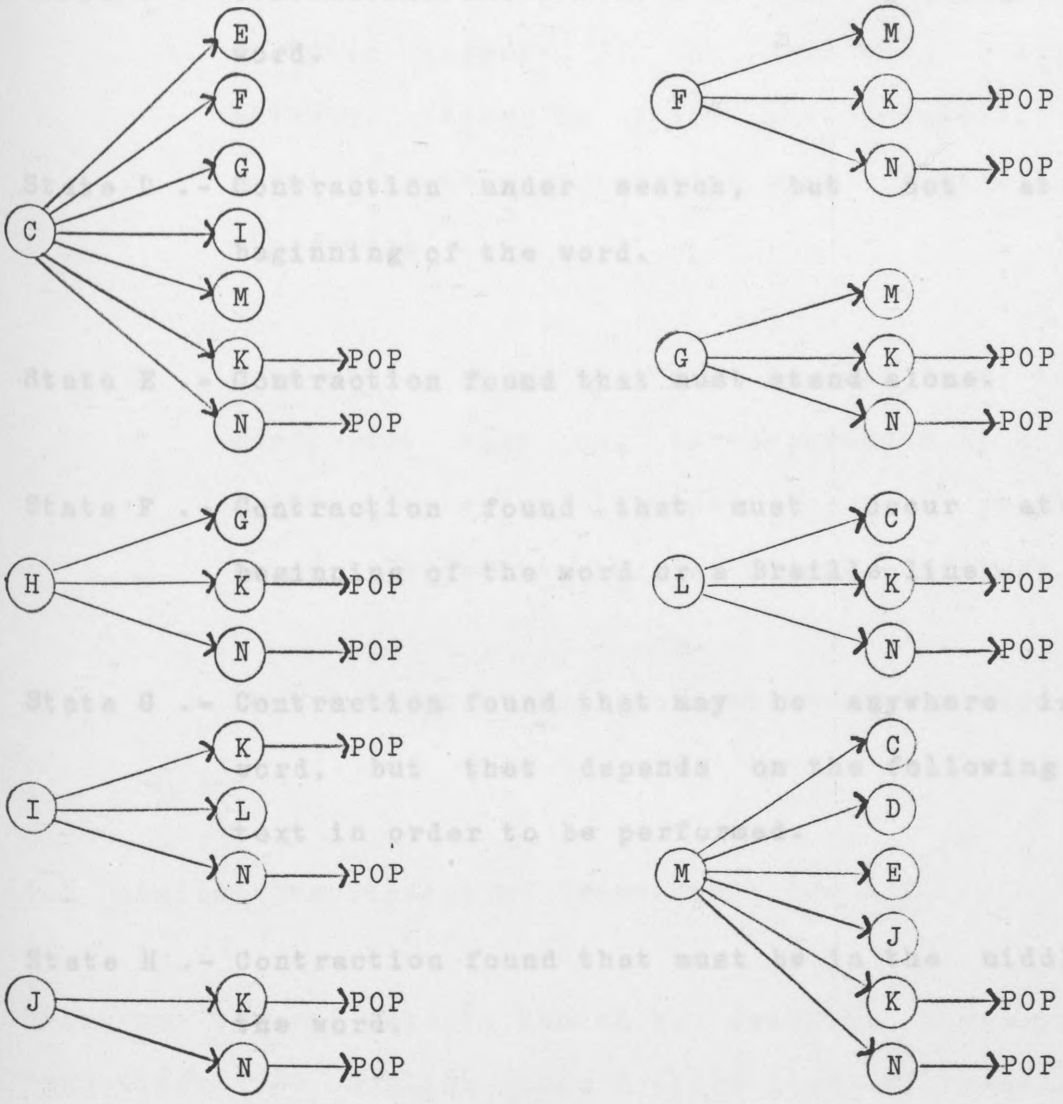


Figure 4.1 (continued)

4.2.2 State Description for English into Braille Grade II Translation

For the sake of completeness of the network presented, a brief description of each of the states comprised therein follows.

State C .- Contraction under search at the beginning of the word.

State D .- Contraction under search, but not at the beginning of the word.

State E .- Contraction found that must stand alone.

State F .- Contraction found that must occur at the beginning of the word or a Braille line.

State G .- Contraction found that may be anywhere in the word, but that depends on the following input text in order to be performed.

State H .- Contraction found that must be in the middle of the word.

State I .- Contraction found that must be used adjoining the word that follows.

State J .- After having found a contraction such as "AND", "FOR", "OF", "THE", "WITH", standing alone, then if there is another of these also standing alone, the space will be contracted.

State K .- A contraction has been found, and nothing may now hinder its performance.

State L .- A contraction that was previously found was not accepted because of the following character; however, there is still the possibility of forming another contraction with the same initial characters.

State M .- Contraction found that may be anywhere in the word, but that can be superseded by a longer contraction which includes the one previously found. The contraction is always performed.

State N .- The contraction under search was not found.

4.3 Detailed Description of Transitions for EBT

There are several aspects needed to describe completely a transition for English into Braille grade II translation; namely,

- a brief description of the transition
- the mathematical productions with which the transitions are associated, together with a glossary of the relevant variables.
- a precise specification of the tests that must be satisfied for the occurrence of the transition.

. the actions that shall be taken if the conditions specified by the tests are satisfied.

Each transition within the network will be described in this way, together with examples that show valid cases for performing the transitions.

Since the tests apply to the state to which the transition is going to be made, the network description will be ordered according to the state reached through the transition (not through the departing state), in the following way: 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'L', 'M', 'K', 'N'. This will allow the grouping of transitions, thus facilitating the description (refer to figure 4.1).

4.3.1 Transition from states 'S', 'C', 'L', 'M' to state 'C'.

1. Description:
   Contraction under search at the beginning of the word.

2. Mathematical production:
   $(q_i, bw, Z\alpha, \{t\}) \vdash ('C', \gamma\alpha, \{a\})$, where:

   q_i - Any of the states: 'S', 'C', 'L' or 'M'.

b - Input character under the input head. the valid input characters can be extracted from the English Braille abbreviation and contraction dictionary shown in appendix 3.

ω - Remaining input text.

Z - Topmost symbol of pushdown stack.

α - Remaining contents of pushdown stack. It is worth noting that the stack contents are shown in reverse order, except when a contraction has already been performed, in which case the contraction is spelled out in the correct order. Contractions are shown in uppercase letters, while non-contracted text in the stack is shown in lowercase letters.

{t} - Set of tests to be satisfied for these transitions to occur. There are general tests for each transition and also specific ones according to the departing state.

General Tests:

1. The translation process status must be at the beginning of the word.

2. There must be possibility of performing a

contraction.

3. The possible contraction must be incomplete.

Specific Tests:

1. Departing state: 'S'

   There is no contraction under search.

2. Departing state: 'C'

   The contraction under search is still possible and incomplete.

3. Departing state: 'L'

   The performance of a contraction has been hindered because of the following input character, but there is the possibility of finding a longer contraction.

4. Departing state: 'M'

   A contraction has been performed, which may form part of a longer and still incomplete contraction or abbreviation.

'C' - State to which the transition reaches.

γ - New topmost symbol of the pushdown stack. It

is worth noting here that the symbols in the pushdown stack are shown in reverse order from the one in which they were pushed, in lowercase letters. When a contraction has been performed, it appears in the correct order and in uppercase letters.

{a} - Set of actions to be performed when the previous conditions are met:

1. Postpone the decision of contracting or not [1].

2. Update the character under search (the one already found) to the one that should appear next in order to be able to contract later on. This is done according to the translation dictionary.

3. Examples of valid transitions [2,3]

---

[1]. The fact of postponing a decision implies that the related information is pushed on to the top of the stack, until enough information is available in order to decide whether or not to contract the input text.

[2]. It is worth mentioning that in this section only the specific transitions related to the cases under consideration are shown in order to not extend excessively. For complete examples refer to the last section of next chapter.

[3]. The stack values shown in the examples are not the real Braille mapping pushed on to the stack. This is done for ease of understanding for the computer-oriented reader. For the Braille-oriented reader, lowercase represents single letters, and uppercase represents contractions already performed.

1. Transition 'S' to 'C'

   ('S', frω, {e}, {t}) ⊢ ('C', rf{e}, {a})

   In this case, since the transition departs from state 'S', there is no contraction under search. Since the pushdown stack is empty, the process status is at beginning of the word. The first two input characters are 'fr', which allow the possibility of contracting (searching for the word 'FRIEND'). Finally, it is obvious that the contraction is not yet complete, so all of the tests required are satisfied and thus the transition to state 'C' occurs.

2. Transition 'C' to 'C'

   ('C', iω, rf{e}, {t}) ⊢ ('C', irf{e}, {a})

   In this case, continuing with the previous example, since 'i' is the next input character and the word 'FRIEND' is being considered, there is still the possibility of contracting, and the contraction is still incomplete. Thus, all the required tests are satisfied and the transition 'C' to 'C' occurs.

3. Transition 'L' to 'C'

   ('L', eω, sti{e}, {t}) ⊢ ('C', esti{e}, {a})

   In this case, the word 'ITS' was found. Had it

been standing alone, it would have been contracted. Since an 'e' followed, the previous contraction was hindered, and the new contraction searched for is 'ITSELF'. Since it is still incomplete, all the tests required are satisfied, and the transition to state 'C' occurs.

4. Transition 'M' to 'C'

('M', rω, THE{e}, {t}) ⊢ ('C', rTHE{e}, {a})

In this case, the word 'THE' has been found and contracted, whatever may come later on. Since the next input character is 'r', there is still the possibility of contracting the word 'THERE'. Since before 'THE', the stack was empty, and the process status was at the beginning of the word. Finally, since the word is still incomplete, the transition to state 'C' occurs.

In any of the previous cases, if one or more of the required conditions are not satisfied, the transition does not occur.

4.3.2 Transitions from states 'S', 'D', 'M' to state 'D'.

1. Description:

   Contraction under search, but not at the beginning of the word.

2. Mathematical Production:

   $(q_i, b\omega, z\alpha, \{t\}) \vdash ('D', \gamma\alpha, \{a\})$, where

   q_i - any of states: 'S', 'D' or 'M'

   b,w,z, have been already defined in the first transition described. From here on, the elements that were previously defined will not be repeated. For their description, refer to section 4.3.1.

   $\{t\}$ - set of tests to be satisfied for these transitions to occur.

   General tests:

   1. The translation process status must not be at the beginning of the word.

   2. There must be the possibility of performing a contraction.

   3. The possible contraction must be

incomplete.

Specific Tests:

1. Departing state: 'S'

   There is no contraction under search.

2. Departing state: 'D'

   The contraction under search is still possible and incomplete.

3. Departing state: 'M'

   A contraction has been performed, which may form part of a longer and still incomplete contraction or abbreviation.

'D' - State to which the transition occurs.

γ - New topmost symbol of the pushdown stack.

{a} - Set of actions to be performed when the previous conditions are met:

1. Postpone the decision to contract or not.
2. Update the character under search (the one already found) to the one that should appear next in order to be able to

contract later on. This is done according to the translation dictionary.

3. Examples of valid transitions.

1. Transition 'S' to 'D'

('S', nω, ad{e}, {t}) ⊢ ('D', nad{e}, {a})

In this case, a contraction not at the beginning of the word is searched starting with 'an' at state 'S'. There is the possibility of finding the contraction 'ANCE', and is thus still incomplete. Since all of the required tests are satisfied, the transition to state 'D' occurs. In chapter 5, where the translation process is described, it will become clear how independent, simultaneous searches can be carried out at different positions in the word.

2. Transition 'D' to 'D'

('D', cω, nad{e}, {t}) ⊢ ('D', cnad{e}, {a})

In this case, continuing with the previous example, since 'c' is the next input character, and the contraction 'ANCE' is being searched, there is still possibility of contracting, and the contraction is still incomplete. The

required tests for the transition 'D' to 'D' to occur are satisfied.

3. Transition 'M' to 'D'

('M', nω, OUc {e}, {t}) ⊢ ('D', nOUc {e}, {a})

In this case, the search process is not at the beginning of the word, as the contraction under search is 'OUNT'. Since 'OU' was contracted, and it may form part of a longer contraction, namely 'OUNT', the transition from 'M' to 'D' occurs.

In any of the previous cases, if one or more of the required conditions are not satisfied, the transition does not occur.

4.3.3 Transitions from states 'S', 'C', 'M' to state 'E'.

1. Description:

Contraction found that must stand alone.

2. Mathematical Production:

$(q_i , b\omega, z\alpha, \{t\}) \vdash ('E', \gamma\alpha , \{a\})$, where [4]

q_i - Any of the states: 'S', 'C' or 'M'

---

[4]. The elements that were already defined will not be repeated. Refer to the previous transitions for their description.

{t} - Set of tests to be satisfied for these transitions to occur.

General Tests:

1. The translation process must be at the beginning of the word.

2. A contraction has been found that must stand alone

Specific Tests:

1. Departing state: 'S'
   There is no contraction under search.

2. Departing state: 'C'
   There is a contraction under search at the beginning of the word.

3. Departing state: 'M'
   A contraction has been performed, which forms part of a longer contraction that has already been found.

{a} - Set of actions to be performed when the previous conditions are met.

1. Postpone the decision of contracting until the next input character establishes whether or not the input word stands alone.

2. Establish that the next input character needs to be a delimiter (punctuation mark or space) for the contraction to be performed.

3. Examples of valid transitions.

1. Transition 'S' to 'E'

('S', doω, {e}, {t}) ⊢ ('E', od{e}, {a})

In this case, there was no contraction under search, and the process status was at the beginning of the word. The word 'DO' has been found, which is a valid Braille contraction that must stand alone. Thus, the transition from state 'S' to 'E' occurs.

2. Transition 'C' to 'E'

('C', rω, efta{e}, {t}) ⊢ ('E', ERtfa{e}, {a})

In this case, there was a contraction under search at the beginning of the word, and the word 'AFTER' has been found. It will depend on

the following input character whether or not the word will be contracted. Anyway, the transition from state 'C' to 'E' takes place.

3. Transition 'M' to 'E'

('M', yω, EVER{e}, {t}) ⊢ ('E', yEVER{e}, {a})

In this case, the contraction for 'EVER' has been performed. As the input character is 'y', the contraction for 'EVERY' could be used. However this depends on whether the next input character is a delimiter or not. The transition from state 'M' to 'E' takes place.

4.3.4 Transition from state 'C' to state 'F'.

1. Description:

Contraction found that must occur at the beginning of the word or a Braille line.

2. Mathematical Production:

('c', bω, Zα, {t}) ⊢ ('F', δα, {a}), where

{t} - Set of tests to be satisfied for this transition to occur.

Tests:

1. Contraction under search at the beginning of the word.

2. The following word must fit on the same line; otherwise the contraction is not performed.

{a} - Set of actions to be performed when the specified conditions are met.

1. Postpone the translation until there is enough information to decide whether or not to contract.

2. Update the character under search, to know what input is required in order to be able to perform the translation.

3. Examples of valid transitions.

Transition 'C' to 'F'

('c', sω, id{e}, {t}) ⊢ ('F', sid{e}, {a})

In this case, there is a contraction under search at the beginning of the word. The contraction

'DIS' will be performed only if the following input text satisfies the required conditions. One of these is, for example, that the following character is not an 'h' (as in diśHes). The word under search may be for example 'DISplay'.

4.3.5 Transition from States 'S', 'C', 'D', 'H' to State 'G'.

1. Description:
Contraction found that may be anywhere in the word but depends on the following input text in order to be performed.

2. Mathematical Production:
$(q_i, b\omega, z\alpha, \{t\}) \vdash ('G', \gamma\alpha, \{a\})$, where

q_i - any of the states 'S', 'C', 'D' or 'H'

$\{t\}$ - set of the tests to be satisfied for these transitions to occur

General Tests:

1. A contraction has been found

2. There is possibility that the following input text hinders its performance.

Specific tests:

In this case, there are no specific tests depending on the origin state. State 'G' may be reached from state 'S' whether or not there was a contraction under search, from state 'C' if the contraction under search was at the beginning of the word, and from state 'D' if it was not at the beginning of the word. The transition from state 'H' to 'G' behaves exactly as the one for state 'D'.

{a} - Set of actions to be performed, when the specified conditions are met.

1. Postpone the decision of contracting, until the next input character establishes whether or not the contraction found is aborted.

2. Update the character under search (already found) to the one that should appear next, in order to decide whether to contract or not, later on. This is done according to the translation dictionary.

3. Examples of valid transitions.

1. Transition 'S' to 'G'

('S', beω, {e}, {t}) ⊢ ('G', eb{e}, {a})

In this case, since the transition departs from state 'S', there was no contraction under search, and the word 'BE' has been found. If the following input character is either 'd', 'r', 'e' or 'a', the word 'BE' will not be allowed to be contracted. However, if another character comes next, the contraction 'BE' will be performed. Thus, the transition from 'S' to 'G' occurs.

2. Transition 'C' to 'G'

('C', eω, no{e}, {t}) ⊢ ('G', eno{e}, {a})

In this case, the word 'ONE' has been found, but its contraction may be hindered if the following input character is 'D', 'N' or 'R', as in 'onER/OUs'. Thus, the transition from 'C' to 'G' occurs.

3. Transition 'D' to 'G'

('D', eω, not{e}, {t}) ⊢ ('G', enot{e}, {a})

This case behaves exactly as the previous one. It is just worth noting that here the stack was

not empty before identifying the contraction 'ONE'. The word being searched for is 'tonéD'.

4. Transition 'H' to 'G'

('H', oω, ffe{e}, {t}) ⊢ ('G', offe{e}, {a})

In this case, the word 'effORt' is being searched for. State 'H' was first reached when 'FF' appeared. Then, as an 'O' was the next input character, there was still the possibility of aborting the 'FF' contraction if an 'r' follows. Thus, the transition from 'H' to 'G' occurs.

4.3.6 Transition from State 'S' to State 'H'.

1. Description:

Contraction found that must be in the middle of the word.

2. Mathematical Production:

('S', bω, zα, {t}) ⊢ ('H', γα, {a}), where

{t} - Set of tests to be satisfied for this transition to occur.

Tests:

1. There must not be a contraction under search.

2. The translation process status must not be at the beginning of the word.

{a} - Set of actions to be performed when the specified conditions are met.

Postpone the decision of contracting, until the next input character establishes whether or not the contraction found is accepted.

3. Examples of valid transitions.

Transition 'S' to 'H'

('S', gω, gar{e}, {t}) ⊢ ('H', ggar{e}, {a})

In this case, there is no contraction under search, and the translation process status is not at the beginning of the word. Since a 'g' was on top of the stack and another 'g' appears in the input stream, the transition from 'S' to 'H' occurs.

4.3.7 Transition from States 'S', 'C' to State 'I'.

1. Description:

   Contraction found that must be used adjoining the word that follows.

2. Mathematical Production:

   $(q_i, b\omega, Z\alpha, \{t\}) \vdash ('I', \gamma\alpha, \{a\})$, where

   q_i - Either of the states 'S' or 'C'.

   $\{t\}$ - Set of tests to be satisfied for this transition to occur.

   General tests:

   1. A contraction has been found.

   2. The translation process status must be at the beginning of the word.

   3. The contraction found must be one of the words: 'TO', 'BY', 'INTO'.

   Specific Tests:

   In this case there are no specific tests governed by the origin state.

{a} - Set of actions to be performed when the specified conditions are met.

Postpone the decision of contracting, until the next input character is read. If it is a space, the contraction is still possible, if the following word fits onto the same Braille line. If the input character is a delimiter or another letter, the possibility of contracting is immediately discarded.

3. Examples of valid transitions.

1. Transition 'S' to 'I'

('S', byω, {e}, {t}) ⊢ ('I', yb{e}, {a})

In this case, since the transition leaves the state 'S', there was no contraction under search. Besides, since the stack is empty, the translation process status is at the beginning of the word. Moreover, the word 'BY' has already been found, and there is the possibility that a space follows, so a contraction may be performed later on.

2. Transition 'C' to 'I'

('c', oω, tIN{e}, {t}) ⊢ ('I', otIN{e}, {t})

This case is very similar to the previous one, the only difference being that there is a contraction under search at the beginning of the word. The word 'INTO' has been found in this case.

4.3.8 Transition from States 'M', 'J' to State 'J'.

1. Description:

   After having found a contraction such as 'A', 'AND', 'FOR', 'OF', 'THE', 'WITH', standing alone, then if there is another such contraction also standing alone, the space between them will be contracted.

2. Mathematical Production:

   $(q_i, b\omega, Z\alpha, \{t\}) \vdash ('J', \gamma\alpha, \{a\})$, where

   q_i - Any of the states 'M' or 'J'.

   b - Pair of input characters that match with any of the two first characters of the specified words.

   Z - The topmost element of the stack which must be a space. This is the one that may be contracted.

α - The rest of the stack should contain at least one of the previously specified contractions.

$\{t\}$ - Set of tests to be satisfied for these transitions to occur.

General Tests:

1. A contraction of the group 'A', 'AND', 'FOR', 'OF', 'THE', 'WITH' has been found, standing alone.

2. A space followed this contraction.

3. There is still the possibility, with the current input character, of finding another contraction of this same group.

Specific Tests:

1. For state 'M' as originating state, a contraction of the specified group has been found, and the current input character is a space.

2. For state 'J' as originating state, a contraction of the previous group has been found, there is a space after it, and the current input character must

match one of the starting characters of the same group of contractions.

{a} - Set of actions to be performed when the specified conditions are met.

1. Postpone the decision of contracting.

2. Update the input character under search, to know the input to be able to perform the translation.

3. Examples of valid transitions.

1. Transition 'M' to 'J'

('M', ' 'ω, AND{e}, {t}) ⊢ ('J', AND{e}, {a})

In this case the word 'AND' has already been contracted, and a space is the following input character. This establishes the possibility of contracting the space, depending on the following input text. For example if the following word is 'FOR' the space between 'AND' and 'FOR' will be contracted. However, if the following word is 'FORm', the space between 'AND' and 'FORm' will not be contracted.

2. Transition 'J' to 'J'

('J', foω, AND{e}, {t}) ⊢

('J', fo' 'AND{e}, {a})

This is a continuation of the previous example. The input characters are 'fo', which match with the expectancies in order to be able to contract the space. of course, whether or not the space is contracted still depends on the following incoming text.

4.3.9 Transition from States 'E', 'I' to state 'L'.

1. Description:

A contraction that was previously found was not accepted because of the following character; however, there is still the possibility of forming another contraction with the same initial characters.

2. Mathematical Production:

$(q_i, b\omega, Z\alpha, \{t\}) \vdash ('L', \delta\alpha, \{a\})$, where

q_i - Any of the states 'E' or 'I'

{t} - Set of tests to be satisfied for this transition to occur.

General Tests:

1. A contraction had been found that should stand alone.

2. The current input character hinders the performance of this contraction.

3. There is the possibility of forming another contraction, which includes the text of the previous contraction as the starting part of the new one.

4. The contraction under search is not yet complete.

Specific Tests:

In this case, it does not matter whether the originating state is 'E' or 'I'. In both cases the contraction under search was not performed, but there is still the possibility of forming another longer contraction.

{a}- Set of actions to be performed when the specified conditions are met.

1. Postpone the decision of contracting until more information is available to

decide whether to contract or not.

2. Get the next contraction from the dictionary (if there is one with the same two initial characters), so that according to the current input character, the right contraction in the dictionary is being accessed.

3. Examples of valid transitions.

1. Transition 'E' to 'L'

('E', nω, ERtfa{e}, {t}) ⊢
('L', nERtfa{e}, {a})

In this case, since the input character was not a space, the contraction for 'AFTER' may not be performed. However, this word may still form part of a longer word that may still be contracted, for example, 'AFTERNOON' or 'AFTERWARD'.

2. Transition 'I' to 'L'

('I', dω, ot{e}, {t}) ⊢ ('L', dot{e}, {a})

In this case the contraction for the word 'TO' is prevented since the input character was not a space. However, since the input character is

a 'd' there is the possibility of performing another contraction, and so, the contraction for 'TODAY' will be searched for, from this point on.

4.3.10 Transition from States 'S', 'C', 'F', 'G', 'M' to State 'M'.

1. Description:

Contraction found that may be anywhere in the word, but that can be superseded by a longer contraction which includes the one previously found. The contraction found is always performed, whether or not the longer contraction is identified later on.

2. Mathematical Production:

$(q_i, b\omega, Z\alpha, \{t\}) \vdash ('M', \delta\alpha, \{a\})$, where

q_i - Any of the states 'S', 'C', 'F', 'G' or 'M'.

$\{t\}$ - Set of tests to be satisfied for this transition to occur.

General Tests:

1. A contraction has been found.

2. The contraction will be performed, though it may be superseded by a longer contraction or abbreviation.

Specific Tests:

1. For originating states 'S', 'C' and 'M', there are no specific tests; the transition to state 'M' is performed.

2. For originating states 'F' and 'G', there are tests to determine whether general exceptions exist. This will be clarified in the examples of the next section. For the state 'F', the translation process status must be at the beginning of the word, while for state 'G' it is not restricted.

{a} - Set of actions to be performed when the specified conditions are met.

1. Perform the contraction found.

2. Delay output until there is enough information to know whether the contraction found forms part of a longer contraction or not.

3. Examples of valid transitions.

1. Transition 'S' to 'M'

   ('S', inω, {e}, {t}) ⊢ ('M', IN{e}, {a})

   In this case, there was no contraction under search, and the translation process status is at the beginning of the word. The word 'IN' has been found, and as the transition from state 'S' to state 'M' occurs, the corresponding translation is performed. However, the text is not yet ready for output, since it may form part of a longer contraction (for example, in the word 'INTO').

2. Transition 'C' to 'M'

   ('c', wω, onk{e}, {t}) ⊢ ('M', KNOW{e}, {a})

   In this case, there was a contraction under search at the beginning of the word. The word 'KNOW' has been found, and as the transition from state 'C' to state 'M' occurs, the corresponding translation is performed, since nothing may now hinder its performance. The translated text is kept for further analysis, since it may form part of a longer contraction (as for example in the word 'KNOWLEDGE').

3. Transition 'F' to 'M'

('F', tω, noc{e}, {t}) ⊢ ('M', tCON{e}, {a})

In this case, since the input letter 't' does not hinder the contraction of 'CON', it is performed, and the parse of the input text continues (the word under search in this example could be 'contain'). Had the input letter been an 'e', the transition would have been hindered, thus providing a way of establishing the rule that contraction 'ONE' has precedence over 'CON'.

4. Transition 'G' to 'M'

('G', sω, eno{e}, {t}) ⊢ ('M', sONE{e}, {a})

In this case, since the input letter was not a 'd', 'r' or 'n', the translation for 'ONE' is performed. This still may form part of a longer contraction ('ONESELF') and thus it is not yet output. Had the input letter been one of the letters previously mentioned, the contraction would not have been performed, showing the precedence of contractions 'ED', 'EN', 'ER' over 'ONE'.

5. Transition 'M' to 'M'

('M', eω, TH{e}, {a}) ⊢ ('M', THE{e}, {a})

In this case, 'TH' had been already contracted, and so the translation state was 'M'. Since the next input letter is an 'e', the contraction for 'THE' in now performed. This is kept for further analysis, since this word may still form part of a longer contraction (for example 'THEIR' or 'THERE').

4.3.11 Transition from States 'S', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'L', 'M' to State 'K'

1. Description:

A contraction has been found, and nothing may now hinder its performance. This is a terminal state, and actually is the only one that performs a contraction and at the same time causes a 'POP' from the translation stacks. This is one of the two emptying conditions that exist within the translator.

2. Mathematical Production:

$(q_i, b\omega, Z\alpha, \{t\}) \vdash ('K', \gamma\alpha, \{a\})$, where

q_i - Any of the states 'S', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'L', or 'M'.

$\{t\}$ - Set of tests to be satisfied for these

transitions to occur.

Specific tests:

For state 'I' it is necessary that the following word fits in the same Braille line. Otherwise the contraction cannot be performed. There are no more specific tests required for the transition to state 'K' to occur.

{a} - Set of actions to be performed when the specified conditions are met.

1. Perform the contraction found, thus emptying the stacks under use.

2. Prepare the translated text to be sent to the output file.

3. Examples of valid transitions.

1. Transition 'S' to 'K'

('S', ghω, {e}, {t}) ⊢ ('K', GH{e}, {a})

In this case the contraction for 'GH' has been found. Regardless of what may come next, these two letters are contracted, since nothing may

now hinder the performance of the contraction. For example, the word under analysis could be 'GHost'. It is worth mentioning that in this and the following cases, the stack contents in the mathematical productions are shown before being emptied by the corresponding transition action. This has been done for ease of representation.

2. Transition 'C' TO 'K'

('C', sω, sorca{e}, {t}) ⊢
('K', ACROSS{e}, {a})

In this case, there was a contraction under search at the beginning of the word, and as the input character 's' completed the abbreviation under search ('ACROSS'), it was performed and prepared for output.

3. Transition 'D' to 'K'

('D', dω, nal{e}, {t}) ⊢ ('K', AND1{e}, {a})

In this case, there was a contraction under search, but not at the beginning of the word. The input character completed the contraction, which was performed and prepared for output.

4. Transition 'E' to 'K'

('E', ' 'ω, sih{e}, {t}) ⊢ ('K', his{e}, {a})

In this case, a contraction has been found, but it is required to stand alone, for it to be performed. As the input character is a space, this condition is satisfied and the transition from 'E' to 'K' occurs.

5. Transition 'F' to 'K'

('F', eω, moc{e}, {t}) ⊢ ('K', eCOM{e}, {a})

In this case, since the input character is an 'e', the word 'COMe' is formed, and thus 'COM' is contracted. This is a delayed decision, since the contraction does not involve the current input character, but because of it, the previous text is contracted.

6. Transition 'G' to 'K'

('G', ' 'ω, ecENh{e}, {t}) ⊢

('K', ENCEh{e}, {a})

In this case, since the input character is a space, the contraction for 'ENCE' is performed. Had the input character been 'd', 'n' or 'r', the contraction would have been prevented.

7. Transition 'H' to 'K'

('H', tω, aeh{e}, {t}) ⊢ ('K', tEAh{e}, {t})

In this case, since the input character is not

a delimiter the contraction 'EA' is performed. This is a delayed decision, since the current input character 't' is not involved in the contraction.

8. Transition 'I' to 'K'

('I', ' 'ω, yb{e}, {t}) ⊢ ('K', BY{e}, {a})

There, since the input character is a space the contraction is performed. Had the input character been another delimiter or a letter, the contraction would have been hindered. Note that this is also a delayed decision.

9. Transition 'J' to 'K'

('J', ' 'ω, FOR' 'AND{e}, {t}) ⊢
('K', FOR/AND{e}, {a})

In this case the space between 'FOR' and 'AND' is contracted. This may be done only after knowing that the current input character is a space, so that it can be guaranteed that both words stand alone.

10. Transition 'L' to 'K'

('L', sω, ediseb{e}, {t}) ⊢
('K', BESIDES{e}, {a})

In this case, the word 'BESIDE' had been

detected, but could not be contracted until it was known whether or not it stood alone. Since the current input character is an 'S', the contraction cannot be performed, but as it may form part of a longer contraction, and in this case it does ('BESIDES'), the transition from 'L' to 'K' takes place.

11. Transition 'M' to 'K'

('M', gω, INs{e}, {t}) ⊢ ('K', INGs{e}, {a})

In this case the contraction for 'IN' has already been performed, but as it may form part of a larger contraction ('ING'), and the input character 'g' provides this, the transition from 'M' to 'K' occurs. The contraction performed is 'ING' in the word 'síNG'.

4.3.12 Transition from States 'S', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'L', 'M' to State 'N'

1. Description:

The contraction under search was not found. This is a terminal state, and actually is the counterpart of state 'K'. This state forbids definitely the peformance of a contraction and at the same time causes a 'POP' from the translation

stacks. This is the other state (together with state 'K') that may cause the emptying of the stacks.

2. Mathematical Production:

$(q_i, b\omega, Z\alpha, \{t\}) \vdash ('N', \gamma\alpha, \{a\})$, where

q_i - Any of the states 'S', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'L', or 'M'.

$\{t\}$ - Set of tests to be satisfied for these transitions to occur.

General Tests:

This actually is the complement of state 'K'. Any tests that are not satisfied are implicitly considered as candidates for having a transition to state 'N'. Thus, a transition to state 'N' occurs whenever a contraction is not allowed.

Specific Tests:

There are no specific tests for these transitions.

3. Examples of valid transitions.

The examples presented in this section are matched one to one to the examples of the previous section. This is done for comparison purposes to show what happens when a contraction is performed, and when it is forbidden.

1. Transition 'S' to 'N'

   ('S', giω, {e}, {t}) $\vdash$ ('N', ig{e}, {a})

   In this case there is no possibility of contracting, since 'gi' is not included in the set of possible contraction beginnings. It is worth mentioning that in this and the following cases, the stack contents in the mathematical productions are again shown before being emptied by the corresponding transition action.

2. Transition 'C' to 'N'

   ('C', nω, orca{e}, {t}) $\vdash$ ('N', norca{e}, {a})

   In this case, the word 'ACROSS' was being considered. The 'n' (to form the word 'acronym')stopped the contraction and the output is prepared.

3. Transition 'D' to 'N'

   ('D', eω, nal{e}, {t}) $\vdash$ ('N', enal{e}, {a})

In this case, the contraction under search was 'AND', as in the word 'láND', but as an 'e' was the next input character, the contraction was prevented and the text was prepared for output.

4. Transition 'E' to 'N'

('E', sω, sih{e}, {t}) ⊢ ('N', ssih{e}, {a})

In this case, the contraction under search was 'HIS'. As it did not stand alone because of the incoming 's', rather than an incoming space, the contraction is forbidden and the text is prepared for output.

5. Transition 'F' to 'N'

('F', dω, oc{e}, {t}) ⊢ ('N', doc{e}, {a}) .

In this case, the contraction under search was 'COM'. Since a 'd' entered as input character instead of an 'm', the contraction is forbidden and the text prepared for output.

6. Transition 'G' to 'N'

('G', dω, ecENirepxe{e}, {t}) ⊢

('N', EDcENirepxe{e}, {a})

In this case, the contraction for 'ENCE' may not be performed, since there is precedence of contraction 'ED' over 'ENCE'. Thus, 'EN' and

'ED' (not ENCE) are contracted and prepared for output.

7. Transition 'H' to 'N'

('H', ' 'ω, aet{e}, {t}) ⊢
('N', ' 'aet{e}, {a})

In this case, the contraction under search is 'EA'. Since it must occur in the middle of the word, and a space is the following input character, the contraction is forbidden and the text is prepared for output.

8. Transition 'I' to 'N'

('I', ';'ω, yb{e}, {t}) ⊢ ('N', ';'yb{e}, {a})

In this case, the contraction under search is 'BY'. However, in order to be contracted, 'BY' must be followed by a space, because it is contracted adjoining the word that follows. Since a semicolon is the following input character the contraction is forbidden and the text is prepared for output.

9. Transition 'J' to 'N'

('J', mω, FOR' 'AND{e}, {t}) ⊢
('N', mFOR' 'AND{e}, {a})

In this case, the space between 'AND' and 'FOR'

was to be contracted if both words stood alone. As an 'm' was the next input character instead of a delimiter, this contraction is forbidden and the text is prepared for output.

10. Transition 'L' to 'N'

('L', sω, ERtfa{e}, {t}) ⊢
('N', sERtfa{e}, {a})

In this case, the word 'AFTER' was to be contracted if it stood alone, or could have formed part of another word ('AFTERNOON' or 'AFTERWARD'). However, since the input character is an 's', the contraction may not be performed and the text is prepared for output.

11. Transition 'M' to 'N'

('M', sω, INs{e}, {t}) ⊢ ('N', sINs{e}, {a})

In this case, the contraction under search was 'ING'. Since the incoming character was an 's', the contraction may not be performed and thus the text is prepared for output.

By now, all of the transitions of English into Grade II Braille have been explained in detail. However, there are many cases that have variants and that there are not

included here, as they would greatly extend this chapter. To this end, it is worth mentioning that with the information of the previous sections and with the aid of the dictionary described in appendix 4, any example could be generated. However, to present examples of the translation of complete words, it is convenient to describe first the translation process as a whole, in order to show the usage of multiple stacks for embedded processing. Complete examples are thus deferred to the end of the next chapter.

Chapter 5

THE TRANSLATION PROCESS

The purpose of this chapter is to give a thorough description of the English into Braille Translation (EBT) process as a whole. Though there is no intention to present here the actual implementation of it, some references to the translation program are given in order to facilitate the process description. The description given is to be considered within the frame of reference established in the previous chapters.

As is well known, a translation process is very similar in its nature to a compilation process. In fact, compilation may be regarded as a specialised type of translation for programming languages. Within this scope, there may be identified several portions that comprise a translator.

Specifically these are:

(1) Lexical analysis
(2) Bookkeeping or symbol table operations
(3) Parsing or syntax analysis
(4) Code generation or translation

For programming languages, there may be still another part,

regarded usually as code optimisation. For the sake of English into Braille Translation (EBT) this part need not be considered. It may also be noted that for EBT, the lexical analysis is trivial, since it is assumed that any English input text is valid (no attempt is made to correct or identify errors within the input text). Thus, the portions that concern the present chapter are:

. Bookkeeping, where a complete description of the data structures used throughout the translation process is presented.

. Parsing, where strings of tokens are examined to determine whether they obey the structural conventions explicit in the syntactic definition of Braille contractions and abbreviations.

. Translation, which actually provides the means for generating the desired output.

It is convenient to mention that though these processes may be presented separately, they are intimately related, and at some points it may even prove difficult to define a border between them.

Initially, in the bookkeeping section (EBT Dictionary Structure) a thorough description is given of the structure defined for the translation dictionary and of the structure

used for input data storage, process and retrieval. In the second section (Translation Process), a detailed description is given on how the translation is performed. This includes the presentation of the complete translation algorithm. Emphasis is given to the methods used for constructing and accessing the English into Braille translation dictionary and to those related to the actual translation performance. Finally, in the last section of this chapter, complete translation examples are presented, according to the detailed description of the translation process previously given. These use the transitions and notation established in chapter 4.

5.1 Description of the Data Structures

It is very important to provide a good definition of the data structures to be used in any programming process. This can never be over-emphasized. Initially, it establishes the frame of reference for the program development and afterwards it controls the program performance. The more effective the data structures of a process are, the more efficient the process often becomes.

For the specific process with which the present thesis is concerned - English into Braille Translation - there are two major aspects that need to be considered: the construction of the translation dictionary, and the structures used for accessing the dictionary and keeping track of the state of

the translation process.

5.1.1 Structure of the EBT Dictionary.

The EBT dictionary is defined according to the information required for performing the translation and to the planned access of this information. The minimal information requirements for performing English into Braille translation are:

. The sequence of English characters to be contracted.

. The transitions defined for contracting
  - at the beginning of the word.
  - not at the beginning of the word.

. The translation for the characters to be contracted or abbreviated.

The information stored in the dictionary will be accessed very frequently, since for each pair of input characters a check must be made to know if the contraction is feasible or not. Thus, a very fast access mechanism is required in order to optimise the searching process. This has been achieved by two means:

. The establishment of a data structure based on binary

trees and linked lists, which provides very fast access to the information required.

. The consideration of speeding-up techniques, such as binary digrams to minimise the number of times the binary tree is searched. These techniques are discussed in Appendix 5.

The data structures used for the English into Braille translation process will be presented next.

Let a tree structure be defined as follows: a tree structure with base type T is either

. An empty structure

. A node of type T with a finite number of associated disjoint tree structures of base type T, called subtrees.

Next, for sake of completeness, some definitions regarding trees will be given; a thorough treatment of this theme may be found in (Wirth, 1976).

It is customary to depict trees upside down and show their roots (see figure 5.1). Thus, the topmost element in a tree is called its root; if an element has no descendants, it is called a terminal element or a 'leaf'; an element which is

not terminal and is not the root is called an interior node. An ordered tree is a tree in which the branches of each node are sequentially arranged, and of particular importance for the present work are the ordered trees of degree 2, or binary trees. A binary tree is defined as a finite set of elements (nodes) which is either empty or consists of a root (node) with two disjoint binary trees called the left and right subtree of the root. A tree is said to be perfectly balanced if for each node the number of nodes in its left and right subtrees differ by at most 1.

Binary trees are frequently used to represent a set of data whose elements are to be retrieved through a unique key. In the specific case for EBT, this key is generated by the first two letters forming an abbreviation or contraction, according to the formula:

```
key:= nchars*(ord(ch1)-ord('A')) + (ord(ch2)-ord('A'))+1;
```

where:

nchars.- number of characters in the alphabet considered (in this case 26).

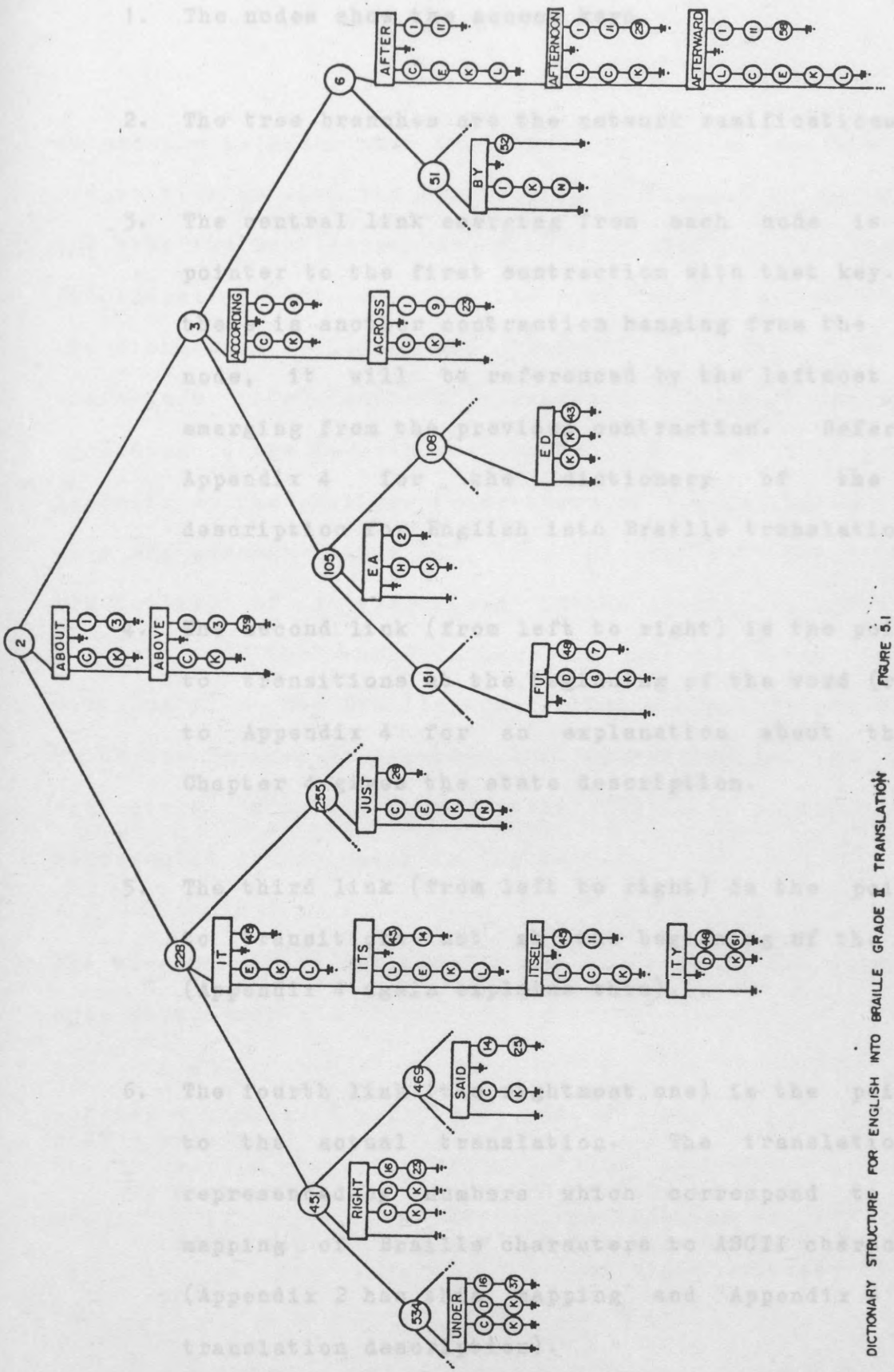
ord.- Pascal function 'ord'.

ch1, ch2.- characters 1 and 2 of the input word.

The use of perfectly balanced trees allows the achievement of high efficiency in information management, since the number of searches needed to reach the desired element is reduced to a minimum.

As was previously mentioned, the data structure required for holding the dictionary information for EBT is based on binary trees and linked lists. This is organised as depicted in figure 5.1. Referring to this figure, the description of the information stored in the EBT dictionary will be given, starting with the tree description, and continuing with the node description together with the linked lists hanging from each node. The pieces of code given in the text are in pseudo-Pascal code.

Dictionary Structure for English into Braille Grade II Translation



DICTIONARY STRUCTURE FOR ENGLISH INTO BRAILLE GRADE II TRANSLATION. FIGURE 5.1

Notes to figure 5.1

1. The nodes show the access keys.

2. The tree branches are the network ramifications.

3. The central link emerging from each node is the pointer to the first contraction with that key. If there is another contraction hanging from the same node, it will be referenced by the leftmost link emerging from the previous contraction. Refer to Appendix 4 for the dictionary of the ATN description for English into Braille translation.

4. The second link (from left to right) is the pointer to transitions at the beginning of the word (refer to Appendix 4 for an explanation about this). Chapter 4 gives the state description.

5. The third link (from left to right) is the pointer to transitions not at the beginning of the word (Appendix 4 again explains this).

6. The fourth link (the rightmost one) is the pointer to the actual translation. The translation is represented by numbers which correspond to the mapping of Braille characters to ASCII characters (Appendix 2 has this mapping and Appendix 4 the translation description).

7. Earth (⏚) means the end of the linked list (pointer to nil).

An example is given next in which it is shown how this figure ties up with the translation dictionary of Appendix 4 and with the Braille representation in ASCII of Appendix 2. Specifically, let us take the word "AFTER" and go through the dictionary to obtain its translation. The first two characters ("a" and "f") generate the key number 6, according to the formula previously given. As it is seen in Appendix 4, the possible transitions at the beginning of the word are through states "C", "E", "K" and "L". There is no possibility of a transition if the process is not at the beginning of the word (after the second slash in the dictionary). The Braille translation characters are 1, 11, which can be seen in Appendix 2 to correspond to the ASCII characters "a", "f", respectively. All this information is represented pictorically in figure 5.1.

The tree structure defined for EBT is based on the following node structure:

```
noderef = ^node; (* pointer to node *)
node = record
       key         : integer;
       contraction : contref; (* pointer to first
                                 contraction *)
       left, right : noderef  (* pointers to left
                                 and right subtrees *)
     end;
```

```
contref = ^contractions (* pointer to contractions *)
contractions =
       record
        ichars     : transiref; (* pointer to input
                                   characters *)
        transibw   : transiref; (* pointer to transitions
                                   at the beginning of
                                   the word *)
        transinbw  : transiref; (* pointers to transitions
                                   not at the beginning
                                   of the word *)
        transl     : translref; (* pointer to translation *)
        nextcontraction : contref  (* pointer to the next
                                      contraction *)
       end;
transiref   = ^transitions (* pointer to transitions *)
transitions =
       record
        state      : char; (* transitions or characters *)
        nexttransi : transiref  (* pointer to next
                                   transition state *)
       end;
translref = ^translations; (* pointer to translations *)
translations =
       record
        translation     : integer; (* first component of
                                      the translation *)
        nexttranslation : translref  (* pointer to next
                                        component of the
                                        translation *)
       end;
```

This structure is represented in figure 5.1.

5.1.2 Structure of the Stacks.

Associated with the dictionary structure used for EBT, there is another data structure very important for the translation process. This is used for the storage of information regarding the state of the translation process. Unlike the dictionary, which once it has been created, is completely static, the stack structure defines stacks which are very dynamic in nature and their contents are continuously changing depending on the input text to be translated. This

structure is the basis for the representation of the pushdown automaton required for the development of the augmented transition networks used for the English into Braille translation process, and is described in the section related to parsing and translation (section 5.2.2).

Thus, the stack structure defined for information storage throughout the translation process is:

```
stackref = ^stack; (* pointer to stack *)
stack =
    record
     level     : integer;    (* stack level *)
     chs       : char;       (* character on top of stack *)
     brreps    : integer;    (* Braille character
                                representation on top of
                                the stack *)
     posatbw   : boolean;    (* position at the beginning
                                of the word or not *)
     depth     : integer;    (* stack depth *)
     presstate : char;       (* present state *)
     acnode    : noderef;    (* pointer to the node
                                accessed last *)
     accont    : contref;    (* pointer to the contraction
                                accessed last *)
     acichars  : transiref;  (* pointer to the last character
                                accessed in a contraction *)
     numichars : integer;    (* number of input characters
                                read *)
     actransi  : transiref;  (* address of the last
                                transition accessed *)
     actransl  : translref;  (* address of the next
                                translation *)
     next      : stackref    (* stack linking *)
    end;
```

The usage of these two data structures (the EBT dictionary and the stack) just defined, will become clear in the next section.

5.2 Detailed Process Description

Perhaps the best way to describe the English into Braille translation process is with the aid of the pseudocode of the first level of refinement of the translation program. Through it, it will be possible to show the inter-relationships of each of the subprocesses that form the EBT process. This pseudocode is presented next.

```
BEGIN  (*  PROGRAM EBT  *)
openfiles;
initialise;
mapbrailletoascii;
IF gradeii THEN
   buildabconttransnetw;

WHILE NOT eof(inp) DO

   BEGIN
   getbraillerep(brrep,inputch);
   IF NOT newparagraph THEN
      formbrword;
   REPEAT
      IF wordcomplete THEN
         formbrline;
      IF linecomplete OR eof(inp) OR newparagraph THEN
         BEGIN
         outputbrailleline;
         preparenextline
         END;
   UNTIL wordfitinline
   END
END.  (*  PROGRAM EBT  *)
```

The program may be separated into two main sections: the dictionary construction, which is a one time process that introduces all the static information for performing English into Braille translation, and the actual parsing and translation process which makes use of the static

information introduced by the previous section and processes the incoming input text, in order to obtain correct Grade II Braille. The process description will be presented according to these sections, although, it may be necessary to make reference to the program listing, which is presented in Appendix 6.

5.2.1 Dictionary Construction

This process comprises two subprocesses, namely the initialisation procedures and the construction of the abbreviation and contraction transition network. The first part handles the opening of files, the initialisation of variables and the mapping of ASCII to Braille. The second part includes the construction of the tree and linked lists required for contractions, transitions and translations.

5.2.1.1 Initialisation Procedures.

1. "Openfiles" is in charge of opening the files required throughout the whole translation process. These are an input file for reading the dictionary information, another input file for the text to be translated, and one output file which is to hold the translated text.

2. "Initialise" gives initial values to variables that

are required throughout the translation process which must have predefined initial values.

3. "Mapbrailletoascii" assigns Braille characters to each of the inkprint ASCII characters. As each Braille cell is formed by six dots (see chapter 2), it is possible to perform this mapping in a one to one way.

5.2.1.2 Construction of Abbreviation and Contraction Transition Network ("buildabconttransnetw")

The translation dictionary is built just once, as it is never updated. It requires fast access, so trees and linked lists are used for the dictionary storage. A detailed description of the construction of the structures used follows.

1. Tree Construction

   Two trees are built for the translation process: the first, for constructing the actual dictionary of abbreviations and contractions, and the second for building an exceptions dictionary to prevent the performance of certain contractions.

   As was mentioned in section 5.1.1, the trees used are perfectly balanced binary trees. In order to achieve this, it is necessary to know the number n

of nodes in the tree. Thus, the rule for equal distribution is best formulated in recursive terms (Knuth, 1973):

- Using one node for the root.
- Generate the left subtree with nl = n div 2 nodes in this way (NL is the number of nodes to the left).
- Generate the right subtree with nr = n - nl - 1 nodes in this way (NR is the number of nodes to the right).

The rule is expressed as the recursive function "tree" that forms part of the translation program. The listing of this function is given in appendix 6, The Program Listing. It is worth noting the simplicity of this section of the program, which is obtained through the use of recursive procedures. It is obvious that recursive algorithms are particularly suitable when a program is to manipulate information whose structure is itself defined recursively.

2. Linked Lists Construction for Contractions, Transitions and Translations.

The procedures within the program that perform these functions are "formcontraction", "formtransition" and "formtranslation". The three of them make use of recursion for the construction

of the dictionary. This is used because, as it is seen in figure 5.1, several contractions may hang from a single node, and each of these has different transition rules (states) and, of course, different translations. As these hang from each contraction, they enable the use of recursion for each of the three cases: contractions, transitions and translations.

It is worth noting that within the formation of contractions procedure, the assignment of values to the binary digrams takes place, in preparation for the speeding-up techniques to be used to avoid unnecessary tree following wherever possible (see Appendix 5, Speeding-up Techniques). Binary digrams are used both, for the contractions and abbreviations dictionary, and for the exceptions dictionary.

5.2.2 The Translation Process

This process performs the actual parsing and translation of English into Braille Grade II. The first part deals with the processing of the input text. The next part is the most important one and deals with the construction of the Braille word; this has embedded processes for searching the dictionary, and for the translation itself. The last part deals with the much simpler output generation. These

processes are next described in detail.

5.2.2.1 Processing of the Input Text

The procedure for processing the input text is "getbraillerep". It's first task is to read the next character from the input stream. Then, some general operations are performed regarding the accepted input character. They are:

1. Every lowercase input character is converted to uppercase. This is done because in English Braille there is no distinction between uppercase and lowercase. There is a "capital sign", but it is not often used in practice.

2. Certain inkprint characters require two Braille cells for their translation. Thus, an auxiliary Braille cell is needed, and is assigned here.

3. There is a special character for indicating a new paragraph, the backslash symbol. Whenever this is the current input character, a new paragraph is generated in the Braille text.

4. There is the possibility of preventing the contraction of any word in English, by the use of a "bypass" character (the at sign "@"). When this is

found in the input text, every character is translated literally, bypassing all the rules for English into Braille Grade II translation, until a space or punctuation mark occurs.

5. In Braille the opening quotation mark is different from the closing quotation mark. This is handled in this part of the program. The first quotation in the input text is always considered to be an opening one, and the one that follows to be a closing one.

5.2.2.2 Construction of a Braille Word

This translator may translate English into either Grade I or Grade II Braille. The translation to Grade I is a simple one to one mapping and so the Braille words are translated with 100% accuracy. However, the process of forming a Braille Grade II word is the most complex part within the translator and the following have to be considered:

1. The management of the Stacks.

2. The performance of the arbitrary tests related to the Augmented Transition Network (ATN) in use.

3. The performance of the arbitrary actions governed by the ATN in use.

Each of these parts will be described in detail in the following pages.

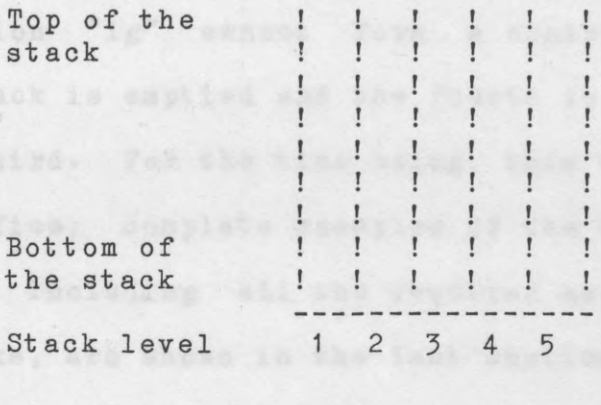
1. The Management of the Stacks.

   In section 5.2.1 the stack structure for the translation process was defined. There follows a description of how this structure is managed throughout the translation process. This actually influences the whole translation process, and so it will overlap with some portions of the translator that will be described later.

   It is well known how a stack behaves; at all times there is access to its topmost part and whatever is to be stored has to be pushed onto its top. Thus, a stack always works with a LIFO (last in, first out) strategy. Formally defined, a stack is a linear list for which all insertions and deletions (and usually all accesses) are made at one end of the list (Knuth, 1973).

   For the specific purposes of the English into Braille Grade II translator, arrays of stacks have been defined in order to be able to hold information at different levels of processing. This is required since a Braille contraction may be

found at any position within the word, and the possibility of contracting at different places frequently occurs, so different stacks must be used for storing information for all such cases. This is represented pictorically in figure 5.2.



Representation of an Array of Stacks

Figure 5.2

In figure 5.2, stack level N ($1 \leq N \leq 5$) is used for searching simultaneously for N Braille contractions at different positions within the same word.

To illustrate how this array of stacks is used during the translation process, take for example the analysis of the word "fright". By the time the third letter is being considered, the word "friend" would be under search with the use of a first stack. A second stack is required for initiating

another analysis, starting with the second letter. In this case, it is for the word "right", which may be contracted anywhere in a word. A third stack is needed to store the "i", since yet another contraction could appear. When the following letter is read ("g"), it is stored initially in a fourth stack; however, since the letter combination "ig" cannot form a contraction, the third stack is emptied and the fourth is displaced to the third. For the time being, this description will suffice; complete examples of the translation process, including all the required management of the stacks, are shown in the last section of this chapter, in section 5.3.

Maintenance of the stacks is performed with two simple procedures: "PUSH" and "POP", which are the only ones allowed to insert or retrieve information from any of the stacks. The assignment of values to the stacks' variables is performed in different places according to the requirements of the translator. The pointers to the tops of the stacks that are in use are held in an array "STACKS". Any access to this array yields a pointer and hence gives a way of reaching a stack, and thus the information it contains.

These stacks constitute the most important part of the translator, since through their use, in

conjunction with the tests and actions performed governed by the dictionary, the augmented transition network, which is the theoretical basis for the construction of this translator, is implemented. The next sections will cover the arbitrary tests and actions realised for the performance of the translation of English into Braille Grade II.

2. Tests performed for the translation of English into Braille Grade II.

According to the philosophy of ATNs, there is the possibility of performing arbitrary tests on the information held on top of the stacks in relation to the last input character. This allows for checking different situations, and so guides the translation process. These checks will be presented next.

1. Checking for numbers.

The translation of numbers from English into Braille introduces some special conditions. Thus, a check is required to translate them correctly. Specifically, the problems in translating numbers arise when they occur as arithmetic operands associated with letters and special symbols as in mathematical expressions.

When letters are used in conjunction with

numbers, every time a change from letter to number or vice versa occurs, a special symbol (called letter or number symbol respectively) must be used in Braille to indicate the change. This is because in Braille the digits (0 - 9) also represent the first 10 letters of the alphabet, so their only distinction is the preceding letter or number symbol.

When numbers (and possibly letters, such as those found in equations) are encountered as arithmetic operands, then the operators "+", "-", "/" and "*" are translated differently from how they are translated when met in ordinary alphabetic text. It is worth noting that special formatting is required when writing numbers in Braille in relation to arithmetic operands, as extra spaces need to be inserted.

2. Checking the Dictionary.

The tests related to checking the presence of words in the dictionary are intimately related with the analysis of transitions. There is a very vague boundary between the tests performed and the actions taken according to the results of the tests. Because of this, the tests performed with the aid of the dictionary will be described briefly, leaving the thorough

analysis to the part where the actions are described, which comes later in this same section.

Dictionary checking is directed mainly at determining whether to contract or not, at the different stages of the process. It is worth mentioning that it is at this point where the use of binary digrams turns out to be extremely useful, since in this way the need to perform tree following at unnecessary times is avoided. If at any of the stages of the process it is determined that there is no possibility of contracting, the corresponding actions taken are to empty the stack and update the state of the process. In a similar way, if it is found that there is the possibility of contracting, the corresponding actions are to push information on to the top of the stack and update the transition state. Finally, if it is determined, according to the dictionary, that a contraction has definitely been found, the corresponding actions for contracting are taken.

3. Checking for Exceptions.

When the formation of a Braille word is completed, there is still a last test to be performed, to see if the translated word lies

within the dictionary of exceptions. For this condition, a positional binary digram is used to speed up the process. If it is shown that it is possible to match an exception from the dictionary to the word just translated, then the exceptions dictionary is accessed. If the translated word is in fact an exception, then it is re-translated according to the exceptions dictionary specifications. If the translated word is not matched with an exception, then it is sent to output unaltered.

3. Actions taken for the translation of English into Braille Grade II.

The actions taken within the translator may be considered of two types: actions related to the dictionary access and actions related to the translation process.

For dictionary access the actions taken are the following:

1. Creation of the key for access.
   This is done through the procedure "getkey", which uses the formula given in section 5.1.1.

2. Location of the key information in the dictionary.

This is done through the procedure "locatekey", which is the one in charge of the tree following. It will either locate the position of the key or inform the caller that it does not exist.

3. Get next contraction ("getnextcont").
   This procedure finds the next contraction with the same key. This occurs when it is determined (through the tests) that the contraction under search was not found, so the next contraction is accessed to see if another contraction can be found.

4. Get position at next contraction ("getposatnextcont").
   In some cases it is necessary not only to get the next contraction, but to remember the exact place where a contraction search was being made. This is done to save time, because the process already initiated for the previous contraction is saved in preparation for the next one. As the next contraction is obtained, the characters forming the current input word are compared, with the characters of the next contraction in the dictionary. If this match fails then there is no possibility of contracting.

The actions related to the translation process are the following:

1. Get next transition ("gettransition").

   The implementation of a contraction or the determination not to contract, occurs according to the transitions defined in the dictionary. This is the procedure that determines the flow within the transition network. Every time text is given as input to the translator, a state is assigned according to its characteristics. Every new input character causes a move from one state to another (or possibly to itself again), until a terminal state is reached. Transitions occur just as presented in chapter 4, where a complete description of the possible transitions between states was given.

   This portion of the translator ultimately serves as the controller for initiating subsequent actions according to the input text. It is this coordinator which establishes when to postpone a decision, when to contract and when not to contract. These actions are discussed next.

2. Postponement of Decisions ("postpone").

   This action is taken whenever there is not enough information to decide whether or not to

contract. It is not a terminal decision that implies an end of a search. The postponement of a decision keeps alive all possibilities, until enough information is available to make a definite decision. The postponement of the decision by no means implies a do-nothing situation. It involves keeping track of the translation by updating the information stored in the stacks. It is a very delicate matter, depending mostly on what is happening in the outer levels of the stacks.

3. Decision not to contract ("donotcontract")

This portion of the translator is in charge of performing all the necessary actions related to the decision of 'not contracting'. This happens when, according to the dictionary (or the binary digrams), there is no possibility of matching a contraction or abbreviation to the input text.

When the decision not to contract is taken, the information stored in the corresponding stack is popped. What is done with this information depends solely on the position of the stack within the array of active stacks. If the stack is the first one, the information in it is stored in an array which eventually will be sent to output. If the stack is not the first

one (the lowest one), the information in it is discarded, since it also exists in the immediately previous stack. Also, if the stack is not the last (the highest one), all stacks after the one under analysis are displaced downwards one position, since this stack was left empty and is not required any more. In this way, it is guaranteed that only the active stacks are alive and that there is automatic garbage collection as the input text is being processed.

4. Decision to contract ("contract").
   This portion of the translator is in charge of performing all the necessary actions related to the decision of contracting. Specifically, these actions are:

   1. Clearing of upper stacks.
      All the stacks above the one where the contraction was found must be cleared. There is, however, an exception to this rule, which consists in taking care not to empty stacks when a delayed contracting decision was taken. A delayed decision means that the contraction was performed as a result of the last input character, but that character is not included in the contraction. When delayed decisions are

made, the outer level stacks will contain all the input text that was not included in the actual contraction.

2. Getting the Translation.

Once that it is decided to contract, it is necessary to read the corresponding translation from the dictionary. At this point the stack position where the contraction was found becomes important. If the contraction is found in the first stack position, then the translation is directly stored in the output array. If this is not the case, then the translation is stored in the corresponding stack, and is propagated downwards into all the preceding stacks. It is important to take into consideration contractions that may form part of yet larger contractions; when this is the case, the contracted text must be kept in the stacks until there is enough information to decide to send it to the output array as it is, or to perform the longer contraction which includes the one previously found, and only then send it to the output array.

5.2.2.3 Output Generation

This portion of the program is in charge of formatting the output text in Braille, according to predefined specifications, and sending it to the output device. The parts that form the output generation are the following:

1. Construction of a Braille line.
   A line length is defined at the beginning of the program, and every Braille line generated must adhere to it. Thus, every time that a Braille word is formed, if it fits into the line, there is no problem and the process continues. If the word does not fit into the line, then the current line is sent to the output stage and after this, the word that did not fit is stored in a new line array that will hold the next line to be output.

2. Output of a Braille line.
   The characters used internally during the translation process are a mapping of ASCII character code to Braille representation (For the description of this mapping, refer to Appendix 2). Thus, the translated text is just a sequence of numbers, which represent bit patterns, that still have to be represented in Braille, i.e., with dots and spaces. The procedure "selectdots" was written for this purpose, and it transfers the Braille line array to another one where the characters are

represented with dots. When this array has been filled, it is sent to the output device. Care is taken at this stage to take into account the page length, which is also defined at the beginning of the program.

5.2.2.4 The Translation Algorithm

Finally, with all the background information previously presented, the translation algorithm is given to show in a simplified way how the translation is performed.

1. A character is read from the input stream.

   1.1 If the character read is a number, there is no possibility of contracting, since all Braille contractions and abbreviations are defined for text. When this happens, any contraction under search is abandoned and all the stacks are emptied. The only thing that must be taken care of is that there exist certain formatting rules in Braille for translating numbers in relation to mathematical signs or letters. The character read is sent to the output array. Go to step 2, for handling the formation of a Braille word.

   1.2 If the character read is a punctuation mark, there is also no further possibility of contracting. However,

this may mean that contractions previously under search can be allowed, since there are contractions that can be performed only if they stand alone or appear at the end of a word. In any case, the appearance of a punctuation mark forces the emptying of stacks, either without contracting or forcing a contraction, depending on what is dictated by the dictionary. The relevant information is stored in the output array. Go to step 2.

1.3 If the character read is a space there are 2 possibilities of action. It may be that according to the dictionary, and of course, to the previous input text, there are no possibilities of contracting the space, and its appearance is equivalent to that of a punctuation mark. If this is not the case (i.e., the space may be contracted), then its appearance is equivalent to that of a letter. This is described next.

1.4 If the character read is a letter, the actual translation from English into Braille Grade II begins. The character read is pushed on top of the first empty stack. The stack level is assigned to the last occupied stack.

1.4.1 If there are no other stacks in use then return to step 1.

1.4.2 If there are other stacks in use, the input character, in combination with previously stored, information is checked against the dictionary.

1.4.2.1 If during this process a contraction is found, all the stacks in use above the stack level being processed (including the present one) are cleared.

- If the stack under process is the first one, the contraction found is transferred to the output array. Go to step 2.

- If the stack under process is not the first one, the contraction found is propagated to all the lower levels. Decrease the stack level under consideration by one, and return to step 1.4.2.

1.4.2.2 If during this process there is the possibility of subsequent contraction, the input character is pushed on to the top of the corresponding stack. This is called postponing the decision of contracting.

- If the stack under process is the first one, return to step 1.

- If the stack under process is not the first one, decrease the stack level under consideration by one and return to step 1.4.2.

1.4.2.3 If during this process there is no possibility of contracting, the corresponding stack is emptied, and all the stacks above this one are displaced one position, eliminating the one for which there is no possibility of contracting.

- If the stack under process is the first one then the information that cannot be contracted is transferred to the output array. Go to step 2.

- If the stack under process is not the first one, decrease the stack level under consideration and return to step 1.4.2.

1.5 If the character read is a special character, the following actions are defined within the translator:

1.5.1 Backslash (\). This character has been defined to cause a new paragraph in the output Braille text. Return to step 1.

1.5.2 At sign (@). This character has been defined as a bypass character, when placed in front of a word that should not be contracted, causes the input word to be copied exactly to the output stream.

1.5.3 Number sign (#). This character has been defined to indicate italicised words, when placed in front of words that are italicised in inkprint.

2. A Braille word is formed.

The output array to which information is transferred during the performance of the translation, is actually an array for storing a Braille Grade II sequence of characters, always delimited by a space, punctuation mark or end of text mark. For the translator such a terminated string is considered to be a Braille word. If the word is complete then go to step 3, otherwise go to step 1.

3. Checking for exceptions.

Once a Braille word has been formed it is necessary to see if it appears in the exception list. If it does not, then go to step 4, otherwise consult the exceptions

dictionary and perform the correct translation. Continue to step 4.

4. A Braille line is formed.

   A Braille line is formed from complete Braille words. If a Braille word does not fit into the current line, it is saved temporarily for the following line. It is important to know if a certain word fits into the current line at translation time, since it affects the translation process. In fact, this may govern whether a certain word is contracted or not. If a line is complete then go to step 5, otherwise return to step 1.

5. Output of a Braille line.

   The newly formed Braille line is sent to an output file. If a word was temporarily saved because it did not fit in that Braille line, it is stored at the beginning of a new Braille line. If there is more input text return to step 1, otherwise send the last Braille line (if there was one) to the output file and finish the translation.

The algorithm presented accounts for the most relevant actions taken within the translator. With this aid, it is possible to generate by hand the translation of whatever input text is entered. This will be demonstrated with the following section.

5.3 Complete Examples of English into Braille Grade II Translation

In this section some examples are presented, in which the translation from English into Braille Grade II is shown step by step, exactly as it is performed by the translation program. Initially, a section for notation will be given. Then, for each example presented, the mathematical development of the translation will be given, according to the notation previously defined. For the first example a very detailed associated description will be included, to aid understanding. For the rest of the examples this description will be reduced to highlight new features.

5.3.1 Notation

Most of this notation has been already included in Chapter 4. However, for sake of completeness of this section, the relevant parts for the examples will be repeated here.

Let $(q_i, b\omega, Z\{e\}, \{t\}) \vdash (q_j, \delta\{e\}, \{a\})$

be a move within the transition network, where:

q_i represents the initial state before the transition. Valid values for it are: S, D, E, F, G, H, I, J, L, M.

b represents the character under the input head. It may be any valid English character.

ω represents the unused portion of the input stream. It may be any valid English text.

Z represents the contents of the stack under consideration. These are always shown in reverse order (as they were pushed onto the stack) and in lowercase letters, except when a contraction has been performed. In this case the contents are shown in the correct order and in uppercase letters.

{e} represents the empty set, that is, the contents of the stack are null.

{t} represents the set of arbitrary tests to be performed in order to accept the transition proposed by the move within the transition network.

⊢ represents the actual move within the transition network and is read "produces".

q_j represents the final state after the transition. Valid values for it are: S, C, D, E, F, G, H, I, J, K, L, M, N.

γ represents the new contents of the stack. It may have either a new character on top of it or a

contraction or abbreviation, if one was found.

{a} represents the set of arbitrary actions to be performed when the transition proposed by the move within the transition network is accepted.

⟨Stack Levels⟩ represent the different levels at which the transitions occur. There are as many levels as necessary for the embedded processes.

⇓⟨text⟩ represents the text being sent to the output array. This text may include both plain characters and Braille contractions or abbreviations.

⇐⟨text⟩ represents text that has been contracted in stacks of a level higher than 1 and that this text is going to be displaced into all the lower levels of analysis, i.e., contractions found in stack level 3 will be displaced to stacks 2 and 1.

/ represents a separation between two consecutive contractions. As Braille contractions are represented in uppercase, when two of them appear together, they could be misinterpreted if they were not properly separated. This is not actually included in the translated text; it is used just for clarity in the translation representation.

⟨eot⟩ represents the end of input text.

5.3.2 Examples of English into Braille Grade II Translation

Example 1: Translation of the word "FRIGHTFULNESS"

The reason for choosing this word is to be able to show within one word, several searching processes, some of them successful and some unsuccessful. The mathematical development of this translation is shown in figure 5.3. There follows a description of how the translation is performed. This will be done, step by step, according to the numbers on the left of figure 5.3.

Description:

1. The first input character ("f") is read and pushed into the first stack. As at least two characters are needed for any contraction, state "S" remains unchanged.

2. The following character ("r") is read from the input stream. Initially, it is pushed on top of the first empty stack (in this case, stack 2), in preparation for subsequent searches. Then, the input character is checked in combination with the character on top of the last non-empty stack (in this case, stack 1), in order to determine if there is the possibility of contracting. In this particular case the word "FRIEND" exists in the contractions and abbreviations dictionary, so the

letter combination "fr" allows the possibility of contracting. Thus, as the dictionary indicates for this case, the state is changed from "S" to "C", indicating that there is a contraction under search at the beginning of the word.

3. The following character ("i") is read from the input stream and is pushed on top of the first empty stack, stack 3, in preparation for subsequent searches. Then, the input character is checked in combination with the character on top of the last non-empty stack (in this case, stack 2), in order to determine whether there is the possibility of contracting. In this particular case the word "right" is in the contractions and abbreviations dictionary, so the letter combination "ri" allows the possibility of contracting. As the search is now not at the beginning of the word, the state is changed from "S" to "D", governed by the dictionary.

The input character is now checked in combination with the topmost element of the next non-empty stack (in this case, stack 1), taking into account the information supplied by the stack's contents regarding transition state and data for direct dictionary access. Since the input character "i" still allows the possibility of contracting the word "friend", this character is pushed on top of

stack 1 and the transition state remains at state "C", indicating a search for a contraction at the beginning of a word.

4. The following character ("g") is read from the input stream. The process proceeds as described in the previous step. Initially it is pushed on top of the first empty stack (stack 4). Then, it is checked in combination with the character on top of the last non-empty stack (stack 3). Since the characters "ig" are not contracted or abbreviated in Braille, the stored information is popped from this stack, leaving it empty. After this has been done, all subsequent stacks are displaced by 1 in order to use the stack that has just been freed. In this case just stack 4 is displaced backwards to stack 3, which was the one freed. The process then continues with stack 2. In this case the dictionary allows a "g" for the contraction under search (right), so it is pushed on top of the stack and the transition state remains at "D".

The process continues with stack 1. At this point the stack information does not allow the possibility of contracting the word "FRIEND", since the input character was a "g" and not an "e". There is an attempt to find another contraction from the dictionary with the same initial characters ("frig"), but as nothing is found, a

transition to state "N" takes place, indicating that there is no possibility of contracting this sequence of characters. Thus, since this is the first stack, the characters that cannot possibly be contracted are sent to the output array. In this particular case, just the letter "f" is sent to this array, since from the "r" onwards, there is still the possibility of contracting. At this point, stack 1 is emptied (by recursive popping of its contents), and the remaining stacks are displaced by 1 in order to use the stack that has just been freed. The second row of step number 4 in figure 5.3 shows the stacks' contents after these actions have been taken.

5. The character ("h") is read from the input stream. Initially it is pushed on top of the first empty stack (stack 3). Then, it is checked in combination with the character on top of the last non-empty stack (stack 2). Since characters "GH" form a contraction, regardless of what may come later on, a transition to state "K" is performed, using the information stored in the dictionary. Since the contraction performed is not in stack 1, it must be displaced downwards to all stacks in which it is present (in this case, it is displaced to stack 1). When this has been done, there is a check to see that the embedded contraction does not hinder the performance of the contraction under

search (in this case the word "RIGHT" is being considered, and the letter combination "GH" allows for this contraction). When a transition to state "K" occurs, all the information stored in higher stack levels must be popped, freeing these stacks. This is because a contraction has been found and cannot be superseded. In this case, stacks 2 and 3 are freed.

6. The next character ("t") is read from the input stream. Initially, it is pushed on top of the first empty stack (stack 2). Then, it is checked in combination with information held in stack 1. The dictionary indicates that a contraction has been found and that nothing may now hinder its performance. This is accomplished by a transition to state "K". Since the contraction "RIGHT" has been found at stack 1, it must be sent to the output array, and all the stacks have to be cleared by popping their contents. As this happens, the transition state is reset to the start state "S".

7. The input character ("f") is read from the input stream and pushed on top of stack 1, which is the first empty stack. The transition state remains at state "S", as there is no contraction under search.

8. Character ("u") is read from the input stream. Initially it is pushed on top of stack 2. Then, it

is checked in combination with the character on top of stack 1 in order to determine if there is the possibility of contracting. Since the process is still not at the beginning of the word, with the aid of the dictionary it is determined that the sequence of letters "ful" may be contracted. The transition state is changed from "S" to "D", indicating that there is a contraction under search which is not at the beginning of a word.

9. The input character ("l") is read. Initially it is pushed on top of stack 3. Then, it is checked for the possibility of forming a contraction with the topmost contents of stack 2. Since there are no possibilities of contractions with starting characters "ul", a transition to state "N" is forced, causing the popping of the contents of stack 2. At this point the contents of stack 3 are transferred to stack 2, leaving stack 3 empty. The input character is then checked in conjunction with the contents of stack 1. Since the sequence of letters "FUL" is detected, a transition from state "D" to state "K" is performed. This causes "FUL" to be sent to the output array, and all the stacks are emptied again. This causes the transition state to be reset to the start state "S".

10. The following input character ("n") is read from the input stream and pushed on top of stack 1. The

transition state remains at "S", since there is no contraction under search. The process of searching for contractions not at the beginning of the word continues, since until now no delimiters (punctuation marks or spaces) have been found.

11. Character "e" is read from the input stream. Initially it is pushed on top of stack 2. Then, it is checked in combination with the topmost element of stack 1. With the aid of the dictionary, it is determined that the word "NECESSARY" could be currently under examination. This is because "nec" could be part of a contraction which is not at the beginning of the word (as in unNECESSARY). The state is changed from "S" to "D".

12. The following input character ("s") is read from the input stream and pushed on top of stack 3. Then, it is checked in combination with the topmost element of stack 2 against the dictionary. Since the sequence "es" does not allow for any contraction, a transition from "S" to "N" occurs, causing a pop of the contents of stack 2. The contents of stack 3 are transferred to stack 2, leaving stack 3 empty. Then, the input character is checked in combination with the information of stack 1. Since this is an "s", the contraction under search is no longer feasible. However, an attempt is made to find in the dictionary another

possibility of contracting with the same initial characters. This search is successful, and now the sequence of letters "NESS" is under consideration. The transition state remains at "D", and procedures "getnextcont" and "getposatnextcont" are used for getting the next contraction and the position in the dictionary for this next contraction.

13. The following input character ("s") is read from the input stream and pushed on top of stack 3. Then, it is checked against the dictionary in conjunction of the topmost element of stack 2, and since "ss" does not allow for the possibility of contracting, this stack is cleared and the contents of stack 3 are transferred to stack 2. Then, the input character is checked in combination with the information in stack 1. The sequence of letters "NESS" is found, so according to the dictionary, a transition to state "K" occurs. This causes "NESS" to be sent to the output array, and all stacks are emptied again.

14. Since there is no more input (the end of text mark has been found), it indicates that the word under scan is complete and that the translated text can be physically sent to output. The translation process finishes.

step	Stack level 1		Stack level 2		Stack level 3		Stack level 4	
1	(S,fω,{e})	⊢ (S,f)						
2	(S,rω,f)	⊢ (C,rf)	(S,rω,{e})	⊢ (S,r)				
3	(C,iω,rf)	⊢ (C,irf)	(S,iω,r)	⊢ (D,ir)	(S,iω,{e})	⊢ (S,i)		
4	(C,gω,irf)	⊢ (N,⇓f,{e})	(D,gω,ir)	⊢ (D,gir)	(S,gω,i)	⊢ (N,{e})	(S,gω,{e})	⊢ (S,g)
	(D,gir)		(S,g)					
5	(D,hω,gir)	⊢ (D,GHir)	(S,hω,g)	⊢ (K,⇐GH,{e})	(S,hω,{e})	⊢ (S,h)		
6	(D,tω,GHir)	⊢ (K,⇓RIGHT,{e})	(S,tω,{e})	⊢ (S,t)				
7	(S,fω,{e})	⊢ (S,f)						
8	(S,uω,f)	⊢ (D,uf)	(S,uω,{e})	⊢ (S,u)				
9	(D,lω,uf)	⊢ (K,⇓FUL,{e})	(S,lω,u)	⊢ (N,{e})	(S,lω,{e})	⊢ (S,l)		
10	(S,nω,{e})	⊢ (S,n)						
11	(S,eω,n)	⊢ (D,en)	(S,eω,{e})	⊢ (S,e)				
12	(D,sω,en)	⊢ (D,sen)	(S,sω,e)	⊢ (N,{e})	(S,sω,{e})	⊢ (S,s)		
13	(D,sω,sen)	⊢ (K,⇓NESS,{e})	(S,sω,s)	⊢ (N,{e})	(S,sω,{e})	⊢ (S,s)		
14	(S,⟨eot⟩,{e})	⊢ (N,{e})						

Example 1: Translation of 'Frightfulness'

Figure 5.3

Example 2: Translation of the phrase "GREETINGS FOR HIM."
The mathematical development of this translation is shown in figure 5.4. There follows a description of how the translation is performed; this, however, will be given in much less detail than the one for example 1. The description will be done step by step according to the numbers on the left of figure 5.4. The steps that are obvious will be grouped together.

Description:

1,2. At stack level 1, the word "GREAT" is under consideration.

3. At stack level 2 the letter combination "re" does not allow for the possibility of contracting, so this stack is cleared. At stack level 1 the word "GREAT" is still being considered.

4. At stack level 2 the letter combination "ee" does not allow for the possibility of contracting, so this stack is cleared. The search at stack level 1 fails, so the sequence of letters "gre" is sent to the output array. This stack is also cleared.

5. At stack level 1, the letter combination "et" does not allow the possibility of contracting, so the letter "e" is sent to the output array.

6,7. At stack level 2, the contraction for "IN" is performed. It is kept there in case it forms part of a longer contraction. This contraction is displaced to stack level 1, but since it does not allow the performance of the contraction under search ("TIME") nor is any other contraction possible, the letter "t" is sent to the output array and the stack is cleared. Stack 2 is displaced downwards to stack 1.

8. At stack level 1 the contraction "ING" is performed and all the stacks are cleared.

9,10. At stack level 1 the end of the word is detected, so there is no further possibility of contracting and the characters "s space " are sent to the output array.

11,12. At stack level 1 the letters "fo" have been accepted; the contraction under search is "FOR".

13,14. At stack level 1 the contraction "FOR" is performed. The information is kept in this same stack to allow for the possibility of contracting it with a space. This is governed by the dictionary.

15. At stack level 1, the sequence "FOR space " is sent to the output array, since the input character "h" did not allow the possibility of contracting that space. The stack is first cleared and then the input character "h" is stored in it, since this character

contributed to a "delayed" decision.

16,17. At stack level 2, the characters "im" do not allow the possibility of contracting, so this stack is cleared. At stack level 1 the word "him" has been found. It is kept in the stack until there is enough information to decide whether to contract or not. This will be done depending on whether the next input character is a punctuation mark or space which allows contraction, or another letter which does not.

18. At stack level 1 a full stop is detected from the input stream. This enables the performance of the contraction and the sequence "HIM." is sent to the output array.

19. The end of text is detected, so the translation process finishes.

step	Stack level 1		Stack level 2		Stack level 3	
1	(S,gω,{e})	⊢ (S,g)				
2	(S,rω,g)	⊢ (C,rg)	(S,rω,{e})	⊢ (S,r)		
3	(C,eω,rg)	⊢ (C,erg)	(S,eω,r)	⊢ (N,{e})	(S,eω,{e})	⊢ (S,e)
	(C,erg)		(S,e)			
4	(C,eω,erg)	⊢ (N,Ψgre,{e})	(S,eω,e)	⊢ (N,{e})	(S,eω,{e})	⊢ (S,e)
	(S,e)					
5	(S,tω,e)	⊢ (N,Ψe,{e})	(S,tω,{e})	⊢ (S,t)		
	(S,t)					
6	(S,iω,t)	⊢ (D,it)	(S,iω,{e})	⊢ (S,i)		
7	(D,nω,it)	⊢ (N,Ψt,{e})	(S,nω,i)	⊢ (M,IN)	(S,nω,{e})	⊢ (S,n)
	(M,IN)					
8	(M,gω,IN)	⊢ (K,ΨING,{e})	(S,gω,{e})	⊢ (S,g)		
9	(S,sω,{e})	⊢ (S,s)				
10	(S,' 'ω,s)	⊢ (N,Ψs' ',{e})	(S,' 'ω,{e})	⊢ (N,{e})		
11	(S,fω,{e})	⊢ (S,f)				

Example 2: Translation of 'Greetings for him.'.

Figure 5.4

step	Stack level 1		Stack level 2		Stack level 3	
12	(S,oω,f)	⊢ (C,of)	(S,oω,{e})	⊢ (S,o)		
13	(C,rω,of) (M,FOR)	⊢ (M,FOR)	(S,rω,o)	⊢ (N,{e})	(S,rω,{e})	⊢ (S,r)
14	(M,' 'ω,FOR)	⊢ (J,' 'FOR)	(S,' 'ω,{e})	⊢ (N,{e})		
15	(J,hω,' 'FOR) (S,h)	⊢ (N,Ψ FOR' ',{e})	(S,hω,{e})	⊢ (S,h)		
16	(S,iω,h)	⊢ (C,ih)	(S,iω,{e})	⊢ (S,i)		
17	(C,mω,ih) (E,mih)	⊢ (E,mih)	(S,mω,i)	⊢ (N,{e})	(S,mω,{e})	⊢ (S,m)
18	(E,'.'ω,mih)	⊢ (K,Ψ HIM'.',{e})	(S,'.'ω,m)	⊢ (N,{e})	(S,'.'ω,{e})	⊢ (N,{e})
19	(S,<eot>,{e})	⊢ (N,{e})				

Example 2: Translation of 'Greetings for him.'.

Figure 5.4 (continued)

Example 3: Translation of the phrase "The sword of the knight."

The mathematical development of this translation is shown in figure 5.5. The description which follows will be given in the same detail as that of example 2.

Description.

1,2. At stack level 1 the contraction "TH" is performed. It is kept in the stack, since it may form part of a longer contraction.

3,4. At stack level 1 the contraction "THE" is performed. It is kept in the stack as the contraction with a space is possible.

5. At stack level 1 the input character "s" forbids the space contraction, so the sequence "THE space " is sent to the output array. This stack is cleared and stack 2 is moved to stack 1.

6. At stack level 1 there is no possibility of contracting with the starting letter combination "sw". The letter "s" is sent to the output array and the stack is cleared. The contents of stack 2 are moved to stack 1.

7,8. At stack level 2 there is no possibility of

contracting with the starting characters "or". This stack is cleared and the contents of stack 3 are moved to stack 2. At stack level 1 the contraction under search is "WORD".

9. At stack level 2 there is no possibility of contracting with starting characters "rd". This stack is cleared. At stack level 1 the contraction for "WORD" is found and performed, and the contraction for "WORD" is sent to the output array.

It is at this point where the exceptions dictionary is used. The word "sword" is included in it, so the contents of the output array are changed using the correct translation stored in the dictionary.

10,11. At stack level 1 the word "OF" is contracted. It is kept in the stack since a space could be contracted with it later on.

12,13. At stack level 1 there is still the possibility of contracting the space.

14. At stack level 2 the contraction "TH" has been found. This is performed and kept in the stack in case there are further possibilities of contracting. The contraction "TH" is displaced to stack 1, where the possibilities of contracting the space remain alive.

15. At stack level 2 the contraction "THE" has been found. It is performed, kept in the stack and displaced to stack 1. At stack level 1 there is still the possibility of contracting the space between "OF" and "THE"; however, it is necessary to wait to see if the following input character is a punctuation mark or a space.

16,17. The following input characters " space " and "k" allow the contraction of the previous space (at stack level 1) and since there is no more possibility of space contraction (because of the input "k"), the text "OF/THE" is sent to the output array without the intervening space. The last space found is sent to the output array after this. Stacks 1 and 2 are cleared and the contents of stack 3 are moved to stack 1.

18,19,20. At stack level 1 the characters "k", "n", "i" are sent to the output array at each of their individual steps. This is because the respective letter combinations "kn", "ni", "ig" do not allow the possibility of contracting. In each case, stack 1 is cleared and the contents of stack 2 are moved to stack 1.

21. At stack level 1 the contraction "GH" is performed and sent to the output array. All the stacks are cleared.

22,23. At stack level 1 a punctuation mark is detected; this hinders the possibility of further contraction, so the stack contents are sent to the output array (these are "t.").

24. The end of text is detected, so the translation process finishes.

step	Stack level 1		Stack level 2		Stack level 3	
1	(S,tω,{e})	⊢ (S,t)				
2	(S,hω,t)	⊢ (M,TH)	(S,hω,{e})	⊢ (S,h)		
	(M,TH)					
3	(M,eω,TH)	⊢ (M,THE)	(S,eω,{e})	⊢ (S,e)		
	(M,THE)					
4	(M,' 'ω,THE)	⊢ (J,' 'THE)	(S,' 'ω,{e})	⊢ (N,{e})		
5	(J,sω,' 'THE)	⊢ (N,⇓THE' ',{e})	(S,sω,{e})	⊢ (S,s)		
	(S,s)					
6	(S,wω,s)	⊢ (N,⇓s,{e})	(S,wω,{e})	⊢ (S,w)		
	(S,w)					
7	(S,oω,w)	⊢ (D,ow)	(S,oω,{e})	⊢ (S,o)		
8	(D,rω,ow)	⊢ (D,row)	(S,rω,o)	⊢ (N,{e})	(S,rω,{e})	⊢ (S,r)
	(D,row)		(S,r)			
9	(D,dω,row)	⊢ (K,⇓WORD,{e})	(S,dω,r)	⊢ (N,{e})	(S,dω,{e})	⊢ (S,d)
	Exception: sWORD ⇒ sword					
10	(S,oω,{e})	⊢ (S,o)				
11	(S,fω,o)	⊢ (M,OF)	(S,fω,{e})	⊢ (S,f)		

Example 3: Translation of 'The sword of the knight.'.

Figure 5.5

step	Stack level 1		Stack level 2		Stack level 3	
12	(M,' 'ω,0F)	⊢ (J,' '0F)	(S,' 'ω,{e})	⊢ (S,d)		
13	(J,tω,' '0F)	⊢ (J,t' '0F)	(S,tω,{e})	⊢ (S,t)		
14	(J,hω,t' '0F)	⊢ (J,TH' '0F)	(S,hω,t)	⊢ (M,TH)	(S,hω,{e})	⊢ (S,h)
	(J,TH' '0F)		(M,TH)			
15	(J,eω,TH' '0F)	⊢ (J,THE' '0F)	(M,eω,TH)	⊢ (M,THE)	(S,eω,{e})	⊢ (S,e)
	(J,THE' '0F)		(M,THE)			
16	(J,' 'ω,THE' '0F)	⊢ (J,' 'THE' '0F)	(M,' 'ω,THE)	⊢ (J,' 'THE)	(S,' 'ω,{e})	⊢ (N,{e})
17	(J,kω,' 'THE' '0F)	⊢ (K,ψ0F/THE' ',{e})	(J,kω,' 'THE)	⊢ (N,{e})	(S,kω,{e})	⊢ (S,k)
	(S,k)					
18	(S,nω,k)	⊢ (N,ψk,{e})	(S,nω,{e})	⊢ (S,n)		
19	(S,iω,n)	⊢ (N,ψn,{e})	(S,iω,{e})	⊢ (S,i)		
20	(S,gω,i)	⊢ (N,ψi,{e})	(S,gω,{e})	⊢ (S,g)		
21	(S,hω,g)	⊢ (K,ψGH,{e})	(S,hω,{e})	⊢ (S,h)		
22	(S,tω,{e})	⊢ (S,t)				
23	(S,'.'ω,t)	⊢ (N, t'.',{e})	(S,'.'ω,{e})	⊢ (N,{e})		
24	(S,<eot>,{e})	⊢ (N,{e})				

Example 3: Translation of 'The sword of the knight.'.

Figure 5.5 (continued)

Chapter 6

PERFORMANCE EVALUATION

This chapter is devoted to the description of the performance characteristics of the translation program developed. The first section discusses general performance characteristics of the translator in an isolated environment. The second section deals with a comparison with existing translators, in order to establish the position of the one implemented within the frame of reference defined by the existing translators.

6.1 General Performance Characteristics

At this moment it is necessary to evaluate the performance of the translator written using the techniques specified in the previous chapters. Five aspects will be considered in this evaluation:

- general program characteristics

- accuracy of the translated text

- speed of the translation (words per minute)

- trainability in relation to new rules or exceptions

- limitations existing in the current implementation.

Each of these is important, but for different purposes, and they will be described next, separately.

6.1.1 General Program Characteristics

Language: PASCAL

Number of lines of the code: 1750 approximately.

Files: Three. Two input files, one for the input text and the other for the dictionaries of rules and their exceptions. The output file holds the translated information.

Memory usage: About 32K bytes of static memory and 3K of dynamic work space.

Dictionary: 189 entries

Exceptions dictionary: 50 entries.

There is flexibility in relation to the definition of the output page size, both for width and length.

The program is very easy to use; the only user interaction necessary when running it is to give the names of the input and output files, and to answer whether or not grade II translation is desired. The instructions to run the program, including input data specifications, are given in Appendix 6.

In relation to portability, the translation program has been run on the following machines;

Computer	Operating system	Pascal
PDP-11/70	RSX-11M	OMSI
HONEYWELL	TSS	ISO
WICAT	UNIX	WICAT
CODATA	UNIX	UCSD
APPLE	UCSD	UCSD

Minor changes were required regarding the input-output procedure "OPENFILES"; the required changes are specified with comments in the program listing (see Appendix 6). Some microcomputer versions of Pascal do not implement dynamic storage management using 'new' and 'dispose'. This difficulty was overcome by the use of 'mark' and 'release'. This is also commented in the program listing in appendix 6. In that respect, the procedure 'POP' would need to be modified to overcome the lack of 'dispose' and the main program body would need changing to use 'mark' and 'release'. These alterations to procedure 'POP', and to the main program are shown in comments where they may be made.

6.1.2 Accuracy of the Translated Text

Several tests have been carried out to determine this. Appendix 7 shows in detail the set of tests performed to evaluate the accuracy of the translated text. It may be seen there that the set of tests is complete and aims to cover all of the possibilities that may arise when dealing with English Braille contractions and abbreviations. Tests are performed to check that a contraction or abbreviation is performed when it ought to be, and that it is not performed when it ought not to be.

Of course that the accuracy achieved with these tests is 100%, since they were used to tune the translator. Certainly this cannot be used as a reliable figure, but tests were made with other texts (see table 6.1), and the accuracy achieved was over 99.5%, the problems detected dealt mostly with accents on foreign words and inner quotation marks.

Text Number	Text Description	Number of Words Translated	Accuracy
1	Ordinary text: a letter to GRD written by the author.	467	100.0%
2	Examples from the Braille Primer for testing proficiency in writing Braille	530	100.0%
3	Further examples from the Braille Primer.	403	99.6%
4	Text extracted from Ray Bradbury's "All Summer in a Day"	525	99.7%
5	Set of tests of appendix 7	461	100.0%
6	Tests performed by the R.N.I.B.	43000	99.6%

Tests for Measuring Accuracy of the Translation

Table 6.1

6.1.3 Speed of the Translation

Several tests were designed to evaluate the speed and behaviour of the translator. The same texts as the ones used for accuracy tests were used for evaluating speed. To increase the number of words available, these texts were replicated several times, monitoring each time how long the translation took. This is shown in Table 6.2. It is worth mentioning that the text description is not given there, only the text number from table 6.1 is referenced.

Text Number	Number of replications of the orig. text	Translation Time (sec.) Grade I	Grade II	Number of words translated	Speed (Words per min.) Grade I	Grade II
1	1	8	11	467	3502	2547
1	2	15	21	934	3736	2669
1	4	29	42	1868	3865	2669
1	8	57	83	3736	3933	2701
1	16	113	166	7472	3967	2701
1	32	226	332	14944	3967	2701
2	1	10	16	530	3180	1988
2	2	19	31	1060	3347	2052
2	4	37	56	2120	3483	2271
2	8	70	106	4240	3634	2400
2	16	138	212	8480	3687	2400
2	32	275	424	16960	3700	2400
5	1	9	14	461	3073	1976
5	2	17	25	922	3254	2213
5	4	32	49	1844	3457	2258
5	8	63	97	3688	3512	2281
5	16	125	192	7376	3540	2305
5	32	249	384	14752	3555	2305
6	1	738	1087	43000	3692	2423

Test for Measuring Speed of the Translation on a PDP-11/70

Table 6.2

Tests on texts numbered 3 and 4 were not considered necessary, as the translation rates reported in table 6.2 are quite consistent. The variation in translation speed is dependent on the average length of the word for the translated text. The longer the average word-length is, the more the translator will delay in the translation. For example, the translation speed for test number 1 is the fastest, as its average word-length is the smallest (5.5 characters per word). The opposite can be said of test number 5, which is not coherent English script. This test is formed from words used to check accuracy, and has no

linking prepositions or articles, which tend to reduce the average word-length of natural text. The word length in this case was 6.4 characters per word.

6.1.4 Trainability of the translator

One of the features of this translator is the possibility of training the translator according either to new rules that may emerge in the future, or to deficiencies which may be observed while testing. This tuning is done through changes in the dictionary of exceptions. These changes include additions to the dictionary, deletions from it, or changes to existing entries. The procedure to perform changes to the dictionaries is given next. Appendix 4 must be completely understood before making any change.

1. Addition to any of the dictionaries

   If the addition introduces a new pair of initial characters, care must be exercised to add one unit to the number appearing at the beginning of the dictionary. This represents the number of nodes in the tree constructed by the translator. The string defining an entry in the dictionary is constructed according to the specifications of appendix 4. In this case the entry must end with a full stop.

   If the addition introduces an entry in which the initial pair of characters already exist, it should

be inserted where it belongs, alphabetically ordered. In this case the number at the beginning of the dictionaries is not altered. Care must be exercised in that if the new entry is the last with that pair of initial characters, then the full stop in the previous entry should be changed from a full stop to a colon, and the current one must be terminated with a full stop. If the new entry is not the last with that pair of initial characters, then the new entry must be terminated with a colon. The string defining the new entry is constructed according to the specifications of appendix 4.

2. Deletions from any of the dictionaries.
   The deletion procedure is similar to the one given in addition. The only variation is that care must be taken to decrease the number of entries in the tree if the deletion removes the only existing entry with that pair of initial characters. Colons and full stops must be treated exactly as described in the previous section.

3. Changes to existing entries in the dictionaries
   The changes to existing entries must be made according to the specifications of appendix 4.

This feature of trainability extends beyond the correction of entries in the dictionaries, to the translation of different languages, just by providing the corresponding

contractions and abbreviations dictionaries, together with their transitions both at the beginning and not at the beginning of the word. The dictionary of exceptions can also be included. Thus, the present translator may actually be considered as a universal language translator, for a language with the same English alphabet.

6.1.5 Limitations existing in the current implementation of the translator.

There are some aspects that have not been included in the current version of the translator. They are:

1. Abbreviations of value and measurement

   When a symbol or a literal abbreviation of value or measurement follows a numeral, the corresponding literal abbreviation or its equivalent, without the abbreviation point, should be placed in Braille before the numeral sign. This is not performed within the translator. At present the input text is translated exactly as it is given.

2. Translation of text

   - The poetry sign is not handled, so there is no provision for indicating that poetry follows.

   - Double italics sign is not implemented. The fact

of not having a double italics sign means that the single italics sign must be given before every italicised word.

- Inner quotation marks are not implemented. Only outer quotation marks are handled by this translator.

- There is no provision for specifying accents for foreign words.

3. Translation of numbers.

Several features are omitted by this translator when translating numbers.

- Fractions and mixed numbers are not included.

- The mathematical separation sign to separate large numbers is not implemented.

- The decimal point sign is not implemented.

- Roman numerals are translated directly, without the preceeding letter sign, which is required in Braille.

4. Hyphenation rules are not provided.

   In the current version of the translator, whenever a word needs to be hyphenated, it is transferred in entirety to the following line.

5. Braille text formatting.

   A new line is given whenever the "paragraph" symbol appears. No centering of titles is provided.

All of these are considered relatively minor aspects that do not hinder the making of a good translator.

6.2 Comparison with Existing Translators

As it was already mentioned in chapter 2, there exist several Grade II Braille translators, for micro, mini and mainframe computers. The literature regarding the description of these translators generally does not include figures for specifying their speed and accuracy. The literature normally restricts itself to specifying general characteristics such as the hardware for which they are designed, the language in which they are written and the general purpose of the translator. For this reason, the only valid performance comparison may be made in relation to DOTSYS III, the system in use at the R.N.I.B. for translating English into Grade II Braille, and for which the R.N.I.B. give figures regarding their translator's performance.

The translator written for this thesis (EBT) was not run on the same computer in which DOTSYS-III is running (a GC-4070), because the R.N.I.B. machine does not support PASCAL. It is, however, comparable to the PDP-11/70 in which the performance evaluation tests for EBT were run.

DOTSYS-III currently translates about 3000 words per minute, while as it is seen in section 6.1, EBT translates between 2400 and 2700 words per minute. It is worth noting that Grade I translation using EBT achieves about 3600 words per minute, which represents the theoretical upper bound for the speed of translation of Grade II Braille in the computer in which the program was tested.

With respect to accuracy, DOTSYS-III has been tuned over several years and now has over 800 exceptions in its dictionary. It achieves near 100% of accuracy, though proof-readers are still used to validate the correctness of the output. EBT has 50 exceptions in its dictionary and an accuracy of over 99.5%. The main problem of the translator, as reported by the R.N.I.B., is with formatting of the Braille output. These results were obtained when translating 43000 input words of which 0.4% were mistranslated.

With respect to maintainability of the program, DOTSYS-III requires changes in the code of the translator each time that a rule has to be changed; with the trainability facility of the translator written, it is possible to change

rules without having to modify the code of the program and recompile it.

Chapter 7

CONCLUSIONS

7.1 Contributions of the Work Developed to the Field of English into Braille Translation.

A new approach for English into Grade II Braille translation has been developed, using augmented transition networks. The translator provides a good approximation to Grade II Braille and is suitable for running in microprocessors. Its speed and accuracy are satisfactory and its flexibility in the specification of rules and exceptions provide ease of use, thus allowing the program to be handled by people who are not experts in either computer science or Braille.

It is hoped that with the increasing facility of being able to automatically translate English into Grade II Braille, more visually handicapped people will have access to printed information, both for employment and leisure activities.

7.2 Possibilities for the Future

There is still much that can be investigated further in relation to Braille translation. One of these matters is for example, the use of this translator for translating other languages into Braille. The use of this translator in

other languages is straightforward and does not require changes to the program, as long as the language to be translated has the same alphabet and the rules for its translation to Grade II Braille are a subset of the ones defined for translating English to Grade II Braille. The dictionary for the specification of rules would need to be constructed in the same way as that described in appendix 4. Exceptions would also be specified in a similar way.

The program is able to handle the situations where a different mapping of ASCII characters to Braille characters is required. This may happen in other language (i.e. Spanish) with letters which have associated accents. However, in this case the program would need to be altered in the procedure 'MAPBRAILLETOASCII' and recompiled. Appendix 2 shows how it could be implemented.

Regarding exceptions in the choice of contractions, this is handled within the code of the program and defined specifically for English into Grade II Braille. If changes in this respect are needed when specifying the Grade II Braille translation rules for another language, recompilation of the program is required. This is one aspect that could be enhanced within the translation program. It should be possible to feed these exceptional choices of contractions as part of the dictionary of exceptions, thus being able to specify the complete operation for another language, without having to recompile the program. Similarly, another aspect that would be

beneficial in relation to the translation of other languages into Grade II Braille, is to provide an editor for creating and maintaining the dictionary of rules and exceptions. At present, one must learn the translation rules and specify them by hand, using a normal text editor, and know how and when a word or part of a word should be contracted. The automation of this would be of great help in creating and maintaining the translation dictionary.

From the operational point of view, the fact of only building dictionaries and being able to translate different languages with the same program may be of great use for a blind person and organisations for the blind, having to deal only with one program and several dictionaries (one for each language) and not with several such programs.

The translator described, running in small micros (as Apples, for example), could serve individually to Blind people both for employment and leisure activities. If larger micros are considered, i.e. those based on the 68000 chip, regional translation centres could be established, housing processor, printer, and up to 16 terminals (with 1984 technology) which could be used for Braille translation. This would diminish significantly the cost of Braille translation on an individual approach basis.

Finally, it is also worth mentioning that there are very important matters that remained untouched in the present thesis, as are Braille music, mathematics and graphics,

which are currently being developed by the R.N.I.B.

Appendix 1

STANDARD ENGLISH BRAILLE(*)

Braille is a system of embossed characters formed by using combinations of six dots, arranged:

1..4
2..5
3..6

The possible combinations of the six dots give 63 simple signs (plus the space, denoted by no dots), which are shown in figure A.1.1 arranged in seven lines. Herein, the black dots represent the raised points of the sign; the dashes serve to show their position in the group of six. The next figure, figure A.1.2, shows the characters used in forming contractions, some Braille compound signs and mathematical signs.

For ordinary purposes there are two grades of Braille: Grade I uncontracted and Grade II contracted.

For Grade I, the only signs used are those underlined in figure A.1.1 for letters, digits, punctuation marks, and special symbols to indicate poetry, numbers, letters, etc. The only compound characters used in Grade I are those

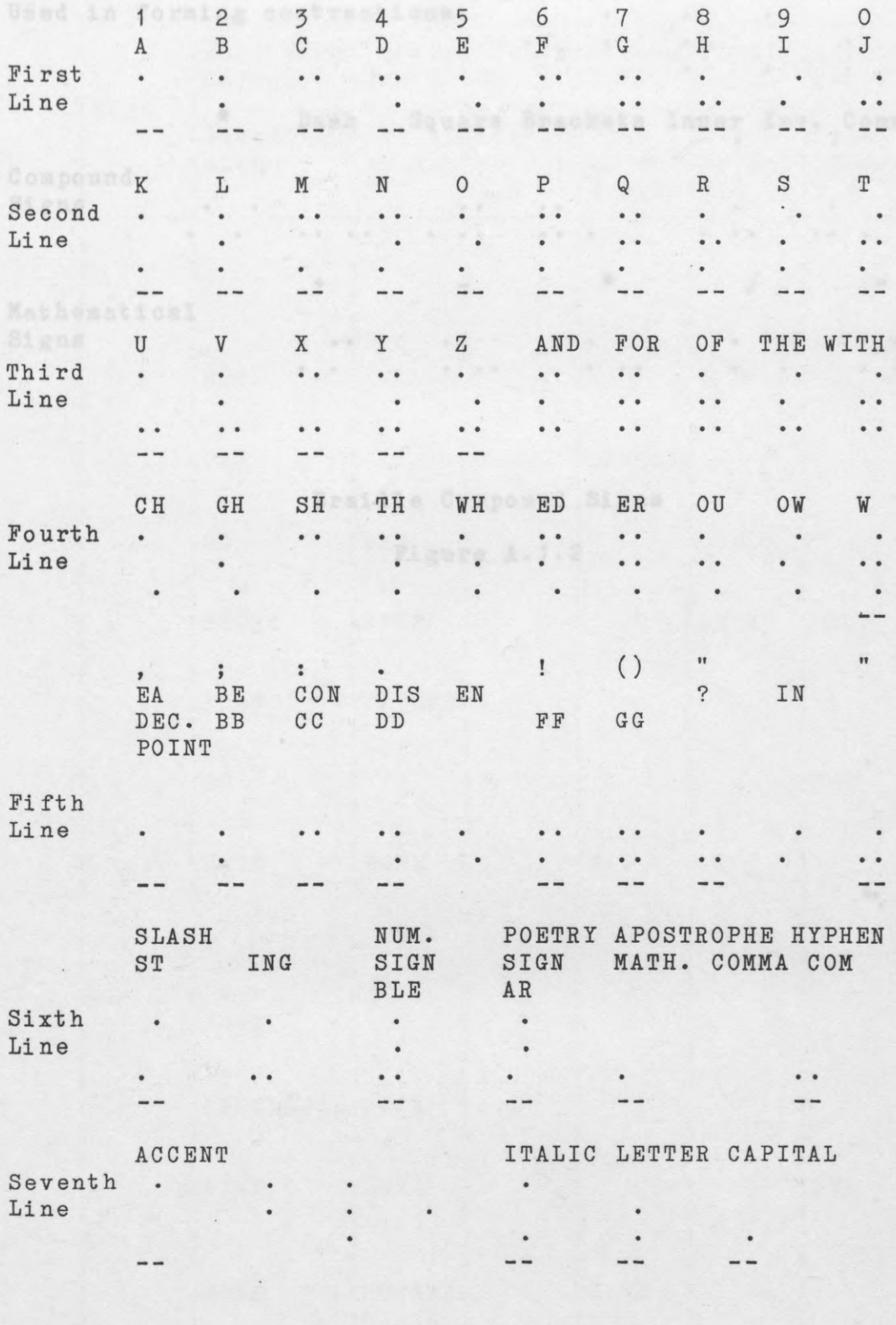
---

(*) This information has been extracted from the Standard English Braille, published by the Royal National Institute for the Blind in 1971.

corresponding to the mathematical signs '+', '-', '*', '/' and '=', and for the symbols '*', dash, square brackets and inner inverted commas (see figure A.1.2).

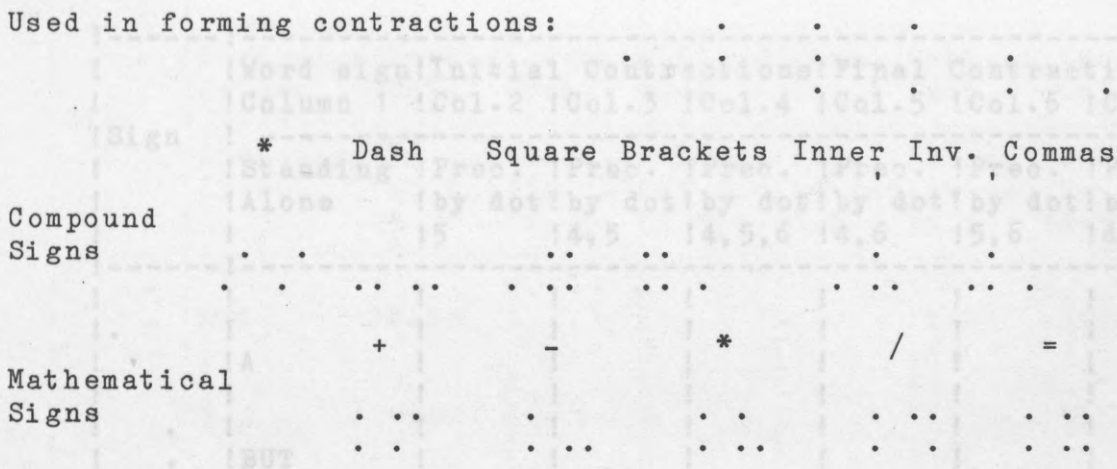
For Grade II the rest of the characters not used in Grade I are used for the most common English contractions. It is worth mentioning that valid rules for Grade I are also valid for Grade II, so Grade II is a superset of Grade I.

Moreover, for Grade II there are other contractions defined by using the letter signs for specifying complete words so that the most common words, standing alone, may be represented by a single letter sign. This is shown in figure A.1.3. In conjunction with the letter sign, some compound contractions are used. Finally, there are some English words (some of the most commonly used), that are abbreviated. The complete list of these is shown in figure A.1.4.



Standard English Braille

Figure A.1.1



Braille Compound Signs

Figure A.1.2

Sign	Word sign Column 1	Initial Contractions Col.2	Col.3	Col.4	Final Contractions Col.5	Col.6	Col.7
	Standing Alone	Prec. by dot 5	Prec. by dot 4,5	Prec. by dot 4,5,6	Prec. by dot 4,6	Prec. by dot 5,6	Prec. by dot 6
.	A						
. .	BUT						
..	CAN			CANNOT			
.. .	DO	DAY			OUND		
. .	EVERY	EVER			ANCE	ENCE	
.. .	FROM	FATHER					
..	GO					ONG	
. ..	HAVE	HERE		HAD			
.	I						
..	JUST						
	KNOWLEDGE	KNOW					
.	LIKE	LORD				FUL	
..	MORE	MOTHER		MANY			

Braille Contractions

Figure A.1.3

Sign	Word sign Column 1	Initial Contractions Col.2	Col.3	Col.4	Final Contractions Col.5	Col.6	Col.7
	Standing Alone	Prec. by dot 5	Prec. by dot 4,5	Prec. by dot 4,5,6	Prec. by dot 4,6	Prec. by dot 5,6	Prec. by dot 6
..							
.	NOT	NAME			SION	TION	ATION
.							
.							
.	O	ONE					
.							
..							
.	PEOPLE	PART					
.							
..							
..	QUITE	QUESTION					
.							
.							
..	RATHER	RIGHT					
.							
.							
.	SO	SOME		SPIRIT	LESS	NESS	
.							
.							
..	THAT	TIME			OUNT	MENT	
.							
.							
	US	UNDER	UPON				
..							
.							
.	VERY						
..							
.							
..	WILL	WORK	WORD	WORLD			
.							
..							
	IT						
..							
..							
.	YOU	YOUNG			ITY	ALLY	
..							
.							
.	AS						
..							
..							
.	AND						
..							

Figure A.1.3 (Continued)

Sign	Word sign Column 1	Initial Contractions Col.2	Col.3	Col.4	Final Contractions Col.5	Col.6	Col.7
	Standing Alone	Prec. by dot 5	Prec. by dot 4,5	Prec. by dot 4,5,6	Prec. by dot 4,6	Prec. by dot 5,6	Prec. by dot 6
..							
..	FOR						
..							
.							
..	OF						
..							
.							
.	THE	THERE	THESE	THEIR			
..							
.							
..	WITH						
..							
.							
	CHILD	CHARACTER					
.							
..							
	SHALL						
.							
..							
.	THIS	THROUGH	THOSE				
.							
.							
.	WHICH	WHERE	WHOSE				
.							
.							
..	OUT	OUGHT					
.							
.	BE						
.							
.	ENOUGH						
.							
..	TO						
.							
..	WERE						
..							
.	HIS						
..							

Figure A.1.3 (Continued)

Sign	Word sign Column 1	Initial Contractions Col.2	Col.3	Col.4	Final Contractions Col.5	Col.6	Col.7
	Standing Alone	Prec. by dot 5	Prec. by dot 4,5	Prec. by dot 4,5,6	Prec. by dot 4,6	Prec. by dot 5,6	Prec. by dot 6
.	IN						
..							
. ...	INTO						
. .							
.	WAS, BY						
..							
.							
	STILL						
.							

Figure A.1.3 (Continued)

English Word	Braille Abbreviation	English Word	Braille Abbreviation
about	ab	herself	hERf
above	abv	him	hm
according	ac	himself	hmf
across	acr	immediate	imm
after	af	its	xs
afternoon	afn	itself	xf
afterward	afw	letter	lr
afterwards	afws	little	ll
again	ag	much	mCH
against	agST	must	mST
almost	alm	myself	myf
already	alr	necessary	nec
also	al	neither	nei
although	alTH	o'clock	o'c
altogether	alt	oneself	ONEf
always	alw	ourselves	OUrvs
because	BEc	paid	pd
before	BEf	perceive	pERcv
behind	BEh	perceiving	pERcvg
below	BEl	perhaps	pERh
beneath	BEn	quick	qk
beside	BEs	receive	rcv
besides	BEss	receiving	rcvg
between	BEt	rejoice	rjc
beyond	BEy	rejoicing	rjcg
blind	bl	said	sd
braille	brl	should	SHd
children	CHn	such	sCH
conceive	CONcv	themselves	THEmvs
conceiving	CONcvg	thyself	THyf
could	cd	today	td
deceive	dcv	together	tgr
deceiving	dcvg	tomorrow	tm
declare	dcl	tonight	tn
declaring	dclg	would	wd
either	ei	your	yr
first	fST	yourself	yrf
friend	fr	yourselves	yrvs
good	gd		
great	grt		

Braille Abbreviations

Figure A.1.4

Appendix 2

MAPPING OF ENGLISH BRAILLE CHARACTERS TO ASCII CHARACTERS

The need to translate English into Braille forces the selection of a character set from which Braille characters will be mapped to English characters. The character set chosen was ASCII, since it is one of the most widely accepted in the field of computing.

As it is well known, ASCII is formed by 128 characters, while Braille has only 64 (see appendix 1). This implies that Braille code will not be able to hold all of the information kept in ASCII. However, this is not even desired, since about 25% of ASCII characters are used for control characters which are not used in Braille. Besides, not all of the English special signs that exist in ASCII have an equivalent in Braille. On the other hand, there are some Braille characters that do not exist in English, as are special symbols to denote capital letters, numeral signs, letter signs, poetry signs and others. These two facts allow a one to one mapping of all the ordinary English characters, as well as the special symbols used in Braille. There are still some empty cells that can be used for special symbols that do not exist in English, but are used in Braille, for contractions and abbreviations of frequently used English character sequences.

It is worth noting that the Braille character set definition is computationally ambiguous, since the same Braille character is used to represent different English (inkprint) characters. This is the case for numbers, for example, which are represented with the same symbols as the first ten letters of the alphabet. This same ambiguity occurs with several different contractions of English letter sequences, which are mapped to the same Braille character depending on the context in which they appear, the context being at the begining, in the middle, or at the end of a word. The opposite case to the one just presented also occurs when translating Braille into English. This happens when different ASCII characters are mapped to one Braille character. Examples of this are the left and right parenthesis, and the opening quotation mark and question mark.

Braille compound characters, which are formed from two Braille cells, represent the ASCII characters: brackets, asterisk and the mathematical symbols +, -, x, /, and =.

All the aspects previously described hinder a totally direct, static mapping from Braille to ASCII, and it must be complemented by a dynamic mapping. However, it is worth noting that even though the dynamic mapping is presented here, it needs to be performed at execution time with the aid of the dictionary, which is given for Braille contractions and abbreviations and were tabulated in appendix 1.

The contents of table A.2.1 are organised in the following way. In the first column, Braille characters are numbered from 0 to 63. The corresponding inkprint character is given in the second column, and the third column contains the ASCII equivalent of this character. When an inkprint counterpart does not exist for a Braille character, its description is given in the second column. If multiple translations are possible for the same Braille character, the alternatives, separated by a comma, are given in the second and third columns.

Finally, it is worth noting that the Braille character's numbering is performed according to the sum of powers of the dots in the Braille cell, numbered as shown in appendix 1. The values are obtained for dots 1-6 by raising 2 to the power 0-5 respectively, thus generating numbers in the range 0 to 63 when summed.

BRAILLE CHARACTER	INKPRINT REPRESENTATION	ASCII EQUIVALENT
0	space	32
1	A, 1	65, 49
2	',', EA	44, none
3	B, 2	66,50
4	apostrophe	39
5	K	75
6	';', BE, BB	59, none, none
7	L	76
8	accent sign	none
9	C, 3	67, 51
10	I, 9	73, 57
11	F, 6	70, 54
12	ST	none
13	M	77
14	S	83
15	P	80
16	used to form compound contractions	none
17	E, 5	69, 48
18	':', CON, CC	58, none, none
19	H, 8	72, 56
20	IN	none
21	O	79
22	'!', FF, TO	33, none, none
23	R	82
24	used to form compound contractions	none
25	D, 4	68, 52
26	J, 0	74, 48
27	G, 7	71, 55
28	AR, poetry sign	none, none
29	N	78
30	T	84
31	Q	81
32	capital sign	none
33	CH	none
34	EN	none
35	GH	none
36	COM, '-'	none, 95
37	U	85
38	'"', '?'	42, 63
39	V	86

Mapping of Braille to ASCII

Table A.2.1

BRAILLE CHARACTER	INKPRINT REPRESENTATION	ASCII EQUIVALENT
40	used to form compound contractions	none
41	SH	none
42	OW	none
43	ED	none
44	ING	none
45	X	88
46	THE	none
47	AND	none
48	used to form compound contractions	none
49	WH	none
50	'.', DIS, DD	46, none, none
51	OU	none
52	'"'	42
53	Z	90
54	'(', ')', GG	40
55	OF	none
56	used to form compund contractions	none
57	TH	none
58	W	87
59	ER	none
60	numeral sign, BLE	none, none
61	Y	89
62	WITH	none
63	FOR	none

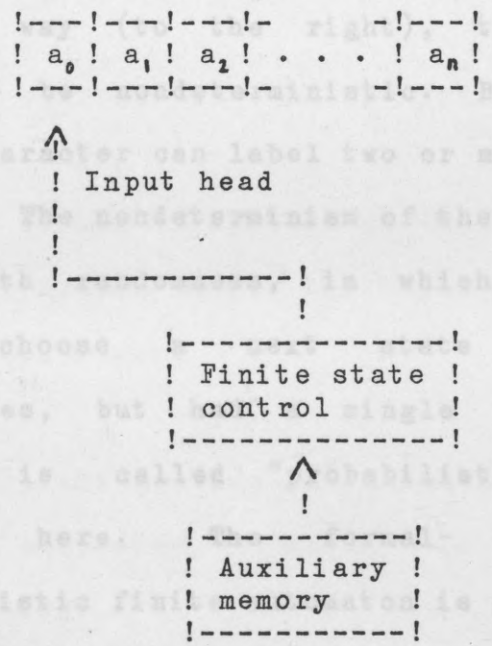
Mapping of Braille to ASCII

Table A.2.1 (continued)

Appendix 3

FINITE AND PUSHDOWN AUTOMATA

One of the most common ways of specifying a language is to define a recogniser for it. A recogniser is formed by 3 elements: an input tape, a finite state control and an auxiliary memory; the type of recogniser varies according to the characteristics of these elements. This is shown pictorically in figure A.3.1. According to this, finite and pushdown automata will be described next.



A recogniser

Figure A.3.1

Finite Automata

Finite automata are used to define the class of regular sets. Other ways of defining these are for example through regular expressions or through the languages generated by right-linear grammars. However, the advantage of using finite automata to define them is that they may be represented by labeled directed graphs, thus allowing easier understanding that with other methods. The class of regular sets defined by a finite automaton is the set of input strings it accepts. A finite automaton is one of the simplest recognisers and ordinarily it consists only of an input tape and a finite control, since its auxiliary memory is null. For this automaton the input head is restricted to move in one way (to the right), the finite control is allowed to be nondeterministic. By this it is meant that the same character can label two or more transitions out of one state. The nondeterminism of the automata should not be confused with randomness, in which the automaton could randomly choose a next state according to fixed probabilities, but had a single existence. Such an automaton is called "probabilistic" and will not be considered here. The formal definition of an nondeterministic finite automaton is given next:

Definition:

A nondeterministic finite automaton (NFA) is a 5-tuple $M=(Q,\Sigma,\delta,q_0,F)$, where:

Q is a finite set of states

Σ is a finite set of permissible input symbols

δ is a mapping from $Q \times \Sigma$ to Q^* which dictates the behaviour of the finite state control; δ is sometimes called the state transtition function

q_0 in Q is the initial state of the finite state control

F is the set of final states (subset of Q)

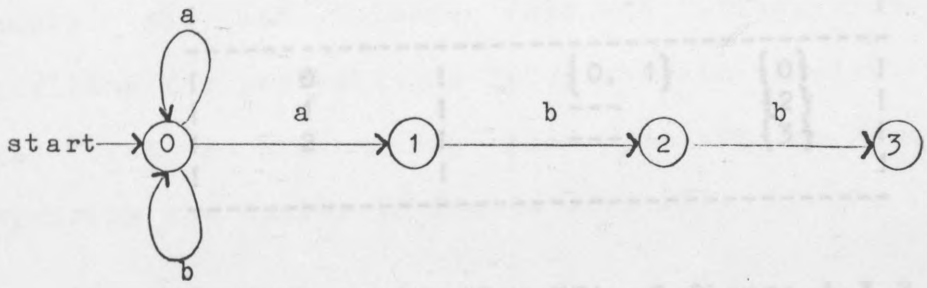
To determine the future behaviour of the automaton, it is necessary to know:

- the current state of the finite control
- the string of symbols on the input tape consisting of the symbol under the input head, and all the symbols to its right.

These two items provide an instantaneous description of the finite automaton, which is normally called its configuration.

An example of an NFA is given next.

Example A.3.1



A Nondeterministic Finite Automaton
Accepting (a|b)*abb

Figure A.3.2

This NFA recognises the language (a|b)*abb. The nodes are called states and the labeled edges are called transitions. One state, (0 in figure A.3.2) is distinguished as the "start state" and one or more states may be distinguished as "accepting states" (or "final states"). In figure A.3.2, the state 3 is a final state.

The transition of an NFA can be conveniently represented in tabular form by means of a transition table. The transition table for the NFA of figure A.3.2 is shown in figure A.3.3. In the transition table, there is a row for each state and a column for each admissible input symbol. The entry for row 'i' and symbol 'a' gives the set of possible next states for state 'i' with input 'a'.

state	input symbol a	input symbol b
0	{0, 1}	{0}
1	---	{2}
2	---	{3}

Transition Table for NFA of figure A.3.2
Figure A.3.3

The NFA accepts an input string 'x' if and only if there is a path from the start state to some accepting state, such that the labels along that path spell out 'x'. The NFA of figure A.3.2 will accept the input strings 'abb', 'aabb', 'babb', 'aaabb', etc. For example, 'aabb' is accepted by the path from state 0 following the edge labeled 'a' to state 0 again, then the states 1, 2 and 3 through edges labeled 'a', 'b' and 'b', respectively. Note that the NFA of figure A.3.2 has more than one transition from state 0 with input 'a'; that is, it may go to state 0 or 1. This situation in which the transition function in multivalued, is the reason why it is hard to simulate an NFA with a computer program. The definition of acceptance establishes that there must be some path labeled by the input string in question leading from the start state to an accepting state.

If there are many paths with the same label string, it is necessary to consider all of them before finding the one that leads to the final state, or to determine that no path

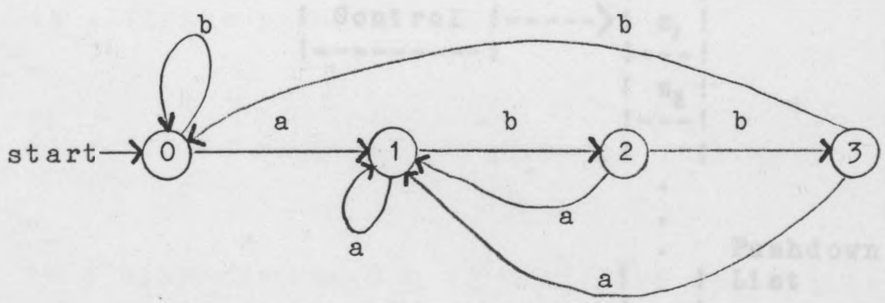
leads to acceptance. For every NFA there is a deterministic version which simulates it in a rather straightforward manner. Aho and Ullman, (Aho and Ullman, 1979), present algorithms for generating a Deterministic Finite Automaton (DFA) from a NFA. They also present an algorithm for minimising the number of states of a DFA.

It is said that a finite automaton is deterministic if:

1. it has no transition on empty input.

2. for each state 's' and input symbol 'a', there is at most one edge labeled 'a' leaving 's'.

Figure A.3.4 shows a DFA accepting the languages (a|b)*abb, which is the same language as the one accepted by the NFA of figure A.3.2.



A Deterministic Finite Automaton
Accepting (a|b)*abb

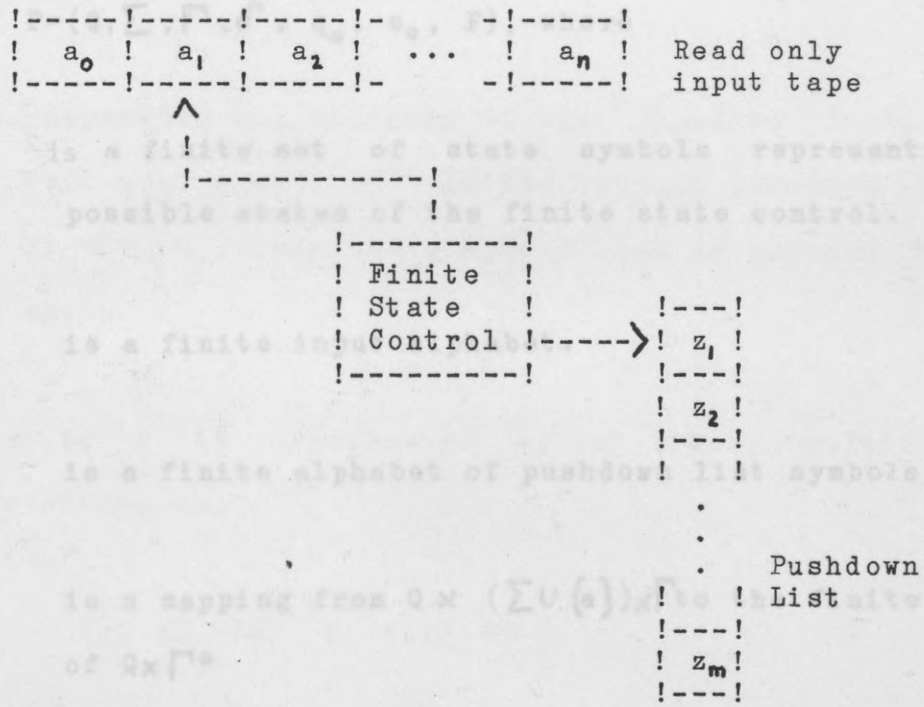
Figure A.3.4

Since there is at most one transition out of any state for

any input symbol, a DFA is easier to simulate by a computer program than an NFA. To simulate a DFA, separate program fragments can be created for each state, each fragment determining the proper transition to make on the current input symbol.

Pushdown automata.

A pushdown automaton is a one-way nondeterministic recogniser whose auxiliary memory consists of one pushdown list, as shown in figure A.3.5.



Pushdown Automaton

Figure A.3.5

This recogniser is the natural model for the syntactic analyser of context free languages. Aho and Ullman (Aho and Ullman, 1972) prove a fundamental result regarding this. They prove that a language is context free if and only if it is accepted by a nondeterministic pushdown automaton. If the context free language is deterministic, then it will be recognised by a deterministic pushdown automaton.

The formal definition of a pushdown automaton is given next.

Definition.

A pushdown automaton (PDA) is a 7-tuple

$P=(Q,\Sigma,\Gamma,\delta, q_0, z_0, F)$, where

Q is a finite set of state symbols representing the possible states of the finite state control.

Σ is a finite input alphabet.

Γ is a finite alphabet of pushdown list symbols.

δ is a mapping from $Q \times (\Sigma \cup \{e\}) \times \Gamma$ to the finite subset of $Q \times \Gamma^*$

q_0 is the initial state of the finite control (subset of Q).

z_0 in Γ is the symbol that appears initially on the pushdown list.

F is the set of final states (subset of Q).

A configuration of P is triple (q, ω, α) in $Q \times \Sigma^* \times \Gamma^*$, where:

q represents the current state of the finite control.

ω represents the unused portion of the input. The first symbol of ω is under the input head. If $\omega = e$, then it is assumed that all of the input tape has been read.

α represents the contents of the pushdown list. The left-most symbol of α is the topmost pushdown symbol. If $\alpha = e$, then the pushdown list is assumed to be empty.

A move by P is represented by a binary relation on configurations as:

$$(q, a\omega, Z\alpha) \vdash (r, \omega, \gamma\alpha)$$

If $\delta(q, a, z)$ contains (r, γ) for any $q \in Q$, $a \in \Sigma \cup \{e\}$, $\omega \in \Sigma^*$, $Z \in \Gamma$, and $\alpha \in \Gamma^*$.

If $a \neq \{e\}$, the previous equation states that if P is in a configuration such that the finite control is in state 'q', the current input symbol is 'a' and the symbol on top of the pushdown list is 'Z', then P may go into a configuration in which the finite control is now in state 'r', the input head has been shifted one square to the right, and the topmost symbol on the pushdown list has been replaced by the string 'γ' of pushdown list symbols. If $\gamma = e$, the pushdown list is said to have been popped. If $a = \{e\}$, then the move is called an e-move. In an e-move, the current input symbol is not taken into consideration, and the input head is not moved. However, the state of the finite control can be changed and the constants of the memory can be adjusted. Note that an e-move can occur even if all of the input has been read; no move is possible if the pushdown list is empty. Some examples of pushdown automata follow.

Example A.3.2

Let us consider Aho and Ullman's design for a pushdown automaton for the language:

$$L = \{ww^r \mid w \in \{a, b\}^+\}$$

This is a language which accepts strings formed by any number of 'a's and 'b's such that the second part of the string matches, in reverse order, its first part.

Let $P = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\})$, where:

(1) $\delta(q_0, a, Z) = \{(q_0, aZ)\}$

(2) $\delta(q_0, b, Z) = \{(q_0, bZ)\}$

(3) $\delta(q_0, a, a) = \{(q_0, aa), (q_1, \{e\})\}$

(4) $\delta(q_0, a, b) = \{(q_0, ab)\}$

(5) $\delta(q_0, b, a) = \{(q_0, ba)\}$

(6) $\delta(q_0, b, b) = \{(q_0, bb), (q_1, \{e\})\}$

(7) $\delta(q_1, a, a) = \{(q_1, \{e\})\}$

(8) $\delta(q_1, b, b) = \{(q_1, \{e\})\}$

(9) $\delta(q_1, e, Z) = \{(q_2, \{e\})\}$

P initially copies some of its input onto its pushdown list, by rules (1), (2), (4) and (5) and the first alternatives of rules (3) and (6). However, P is nondeterministic. Anytime it wishes, as long as its current input matches the top of the pushdown list, it may enter state q_1, and begin matching its pushdown list against the input. The second alternative of rules (3) and (6) represents this choice, and the matching continues by rules (7) and (8). Note that if P ever fails to find a match, then this instance of P "dies". However, since P is nondeterministic, it attempts all possible moves. If any choice causes P to expose the 'Z' on its pushdown list, then by rule (9) 'Z' is erased and state q_2 entered. Thus, P accepts if and only if all matches are made.

For example, with the input string 'abba', P can make the

following sequences of moves:

(1) $(q_0, abba, Z) \vdash (q_0, bba, aZ)$

$\vdash (q_0, ba, baZ)$

$\vdash (q_0, a, bbaZ)$

$\vdash (q_0, \{e\}, abbaZ)$

(2) $(q_0, abba, Z) \vdash (q_0, bba, aZ)$

$\vdash (q_0, ba, baZ)$

$\vdash (q_1, a, aZ)$

$\vdash (q_1, \{e\}, Z)$

$\vdash (q_2, \{e\}, \{e\})$

Since the sequence (Z) ends in final state q_2, P accepts the input string 'abba'. It should be mentioned here that these are not the only sequences of recognising moves.

The pushdown automaton of this example clearly brings out the nondeterministic nature of a PDA. From any configuration of the form $(q_0, a\omega, a\alpha)$ it is possible for P to make one of two moves: either push another 'a' on the pushdown list or pop the 'a' from the top of the pushdown list.

From this, it should be emphasized that although a nondeterministic pushdown automaton may provide a convenient abstract definition of a language, the device must be deterministically simulated to be realised in practice.

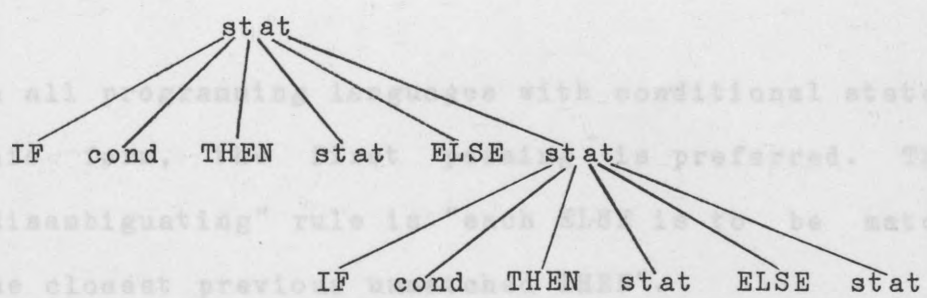
This will be illustrated through an example related to the syntax of programming languages, most of whose constructs can be described by context-free languages which are themselves recognised by pushdown automata.

Example A.3.3

The following grammar fragment generates conditional statements:

```
stat --> IF cond THEN stat
       | IF cond THEN stat ELSE stat
       | other-stat
```

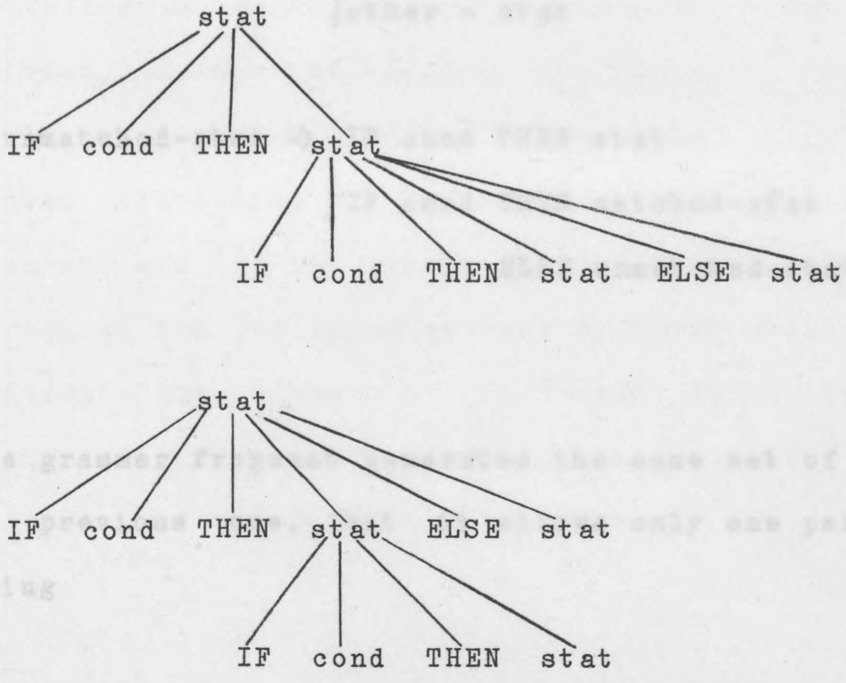
Thus the string
'IF cond1 THEN stat1 ELSE IF cond2 THEN stat2 ELSE stat3'
has the parse tree shown in figure A.3.6



A parse tree

Figure A.3.6

This grammar, however, is ambiguous, since the string 'IF cond1 THEN IF cond2 THEN stat1 ELSE stat2' has the two parse trees shown if figure A.3.7



Two parse trees for an Ambiguous Sentence

Figure A.3.7

In all programming languages with conditional statements of this form, the first parsing is preferred. The general "disambiguating" rule is "each ELSE is to be matched with the closest previous unmatched THEN".

A method of incorporating this "disambiguating" rule into the grammar fragment of example A.3.3 follows.

```
stat ----------> matched-stat
                |unmatched-stat

matched-stat --> IF cond THEN matched-stat
                             ELSE matched-stat
                |other - stat

unmatched-stat -> IF cond THEN stat
                  |IF cond THEN matched-stat
                               ELSE unmatched-stat
```

This grammar fragment generates the same set of strings as the previous one, but it allows only one parsing for the string

"IF cond1 THEN IF cond2 THEN stat1 ELSE stat2",

namely the one that associates the ELSE with the previous unmatched THEN.

Appendix 4

DICTIONARY DESCRIPTION

There follows a description of the notation used for the definition of the dictionary for English into Braille Grade II translation. After this, the dictionary is presented, including (a) the contractions and abbreviations dictionary, and (b) the exceptions dictionary. The numbers appearing at the beginning of each of these sections (89 and 29) indicate the number of different first two letters existing for each case. This is used for efficient construction of the required look-up trees.

General Comments.

1. Slashes are delimiters among different types of items. Specifically, they are used to separate contractions or abbreviations in English from transitions at the beginning of the word, these from transitions not at the beginning of the word, and these from the translation representation.

2. Commas are delimiters within elements of the same type (i.e. to separate one transition from another).

3. Colons indicate that another contraction or abbreviation follows with the same two starting characters.

4. Full Stops indicate that no more contractions or abbreviations that start with the same two characters follow.

Specific Description (Dictionary of Contractions and Abbreviations).

1. Before the first slash, is the contraction or abbreviation to be contracted.

2. After the first slash, is the state transition if the input text matches the dictionary, and if the translation process is at the beginning of the word.

3. After the second slash, is the state transition if the input text matches the dictionary, and if the translation process is not at the beginning of the word.

4. After the third slash, is the actual translation for the contraction or abbreviation under consideration.

5. After the fourth slash, is either a colon or a period, to differentiate whether there are any more contractions starting with the same two initial characters, or not.

Specific Description (Dictionary of Exceptions).

1. Before the first slash, is the word to be considered an exception.

2. After the first slash, is the Braille representation with which the exception word is to be translated by a program.

3. The fields after the second and third slashes are unused.

4. After the fourth slash, is the correct translation for the exception under consideration.

5. After the fifth slash, is either a colon or a period, to differentiate whether there are any more exceptions starting with the same two initial characters, or not.

The dictionary for EBT is shown next in table A.4.1, and the dictionary for exceptions is shown in table A.4.2. It is

worth noting that by specifying exceptions in this dictionary all derived words from the root word will also be considered as exceptions. However, if this is not desired, it is necessary to give a null after the specified part of the word (in its Braille representation), which needs to be considered as an exception, but which is not a root. This is shown in the exceptions dictionary with the word "disc".

For cases where the space is contracted and this is not desired because of the context in which it occurs (i.e., when "by was" appear together, the space should not be contracted). In this special case, the word that should be included as an exception is "was", to prevent the space contraction of being performed. This is exemplified in the exceptions dictionary.

```
! 89,                                          !
! ABOUT/C,K/D,K/1,3/:                           !
! ABOVE/C,K//1,3,39/.                           !
! ACCORDING/C,K//1,9/:                          !
! ACROSS/C,K//1,9,23/.                          !
! AFTER/C,E,K,L/D,K/1,11/:                      !
! AFTERNOON/L,C,K//1,11,29/:                    !
! AFTERWARD/L,C,E,K,L//1,11,58/:                !
! AFTERWARDS/L,K//1,11,58,14/.                  !
! AGAIN/C,E,K,L//1,27/:                         !
! AGAINST/L,C,K//1,27,12/.                      !
! ALLY//D,K/32,61/:                             !
! ALMOST/C,K//1,7,13/:                          !
! ALREADY/C,K//1,7,23/:                         !
! ALSO/C,K//1,7/:                               !
! ALTHOUGH/C,K//1,7,57/:                        !
! ALTOGETHER/C,K//1,7,30/:                      !
! ALWAYS/C,K//1,7,58/.                          !
! ANCE//D,K/40,17/:                             !
! AND/C,M,J,K/D,K/47/.                          !
! AR/K/K/28/.                                   !
! AS/E,K,N//53/.                                !
! ATION//D,K/32,29/.                            !
! BB//H,G,K/6/.                                 !
! BE/G,M//6/:                                   !
! BECAUSE/M,C,K//6,9/:                          !
! BEFORE/M,C,K//6,11/:                          !
! BEHIND/M,C,K//6,19/:                          !
! BELOW/M,C,K//6,7/:                            !
! BESIDE/M,C,E,K,L//6,14/:                      !
! BESIDES/L,K//6,14,14/:                        !
! BETWEEN/M,C,K//6,30/:                         !
! BEYOND/M,C,K//6,61/.                          !
! BLE//D,K/60/:                                 !
! BLIND/C,E,K,N//3,7/.                          !
! BRAILLE/C,E,K,N//3,23,7/.                     !
! BUT/C,E,K,N//3/.                              !
! BY/I,K,N//52/.                                !
! CAN/C,E,K,L//9/:                              !
! CANNOT/L,C,K//56,9/.                          !
! CC//H,K/18/.                                  !
! CH/M/M/33/:                                   !
! CHARACTER/M,C,K/M,D,K/16,33/:                 !
! CHILD/M,C,E,K,L//33/:                         !
! CHILDREN/L,C,K//33,29/.                       !
! COM/C,F,K//36/:                               !
! CON/C,F,M//18/:                               !
! CONCEIVE/M,C,K//18,9,39/:                     !
! CONCEIVING/C,K//18,9,39,27/:                  !
```

Dictionary of abbreviations and contractions for English into Braille Grade II Translation

Table A.4.1

```
! COULD/C,K//9,25/.                    !
! DAY/C,K/D,K/16,25/.                   !
! DD//H,G,K/50/.                        !
! DECEIVE/C,K//25,9,39/:                !
! DECEIVING/C,K//25,9,39,27/:           !
! DECLARE/C,K//25,9,7/:                 !
! DECLARING/C,K//25,9,7,27/.            !
! DIS/C,F,K//50/.                       !
! DO/E,K,N//25/.                        !
! EA//K/2/.                             !
! ED/K/K/43/.                           !
! EITHER/C,K//17,10/.                   !
! EN/M/M/34/:                           !
! ENCE//M,D,G,K/48,17/:                 !
! ENOUGH/M,C,E,K,N//34/.                !
! ER/K/K/59/.                           !
! EVER/C,M/D,K/16,17/:                  !
! EVERY/M,E,K,N//17/.                   !
! FATHER/C,K//16,11/.                   !
! FF//H,G,K/22/.                        !
! FIRST/C,K//11,12/.                    !
! FOR/C,M,J,K/D,K/63/.                  !
! FRIEND/C,K//11,23/:                   !
! FROM/C,E,K,N//11/.                    !
! FUL//D,G,K/48,7/.                     !
! GG//H,K/54/.                          !
! GH/K/K/35/.                           !
! GO/E,K,L//27/:                        !
! GOOD/L,C,K//27,25/.                   !
! GREAT/C,K//27,23,30/.                 !
! HAD/C,E,K,N//56,19/:                  !
! HAVE/C,E,K,N//19/.                    !
! HERE/C,G,K/D,K/16,19/:                !
! HERSELF/C,K//19,59,11/.               !
! HIM/C,E,K,L//19,13/:                  !
! HIMSELF/L,C,K//19,13,11/:             !
! HIS/C,E,K,N//38/.                     !
! IMMEDIATE/C,K//10,13,13/.             !
! IN/M/M/20/:                           !
! ING//M,K/44/:                         !
! INTO/M,C,I,K,N//20,22/.               !
! IT/E,K,L//45/:                        !
! ITS/L,E,K,L//45,14/:                  !
! ITSELF/L,C,K//45,11/:                 !
! ITY//D,K/48,61/.                      !
! JUST/C,E,K,N//26/.                    !
! KNOW/C,M/D,K/16,5/:                   !
! KNOWLEDGE/M,C,E,K,N//5/.              !
! LESS//D,K/40,14/:                     !
! LETTER/C,K//7,23/.                    !
! LIKE/C,E,K,N//7/:                     !
! LITTLE/C,K//7,7/.                     !
! LORD/C,K/D,K/16,7/.                   !
```

Table A.4.1 (continued)

```
! MANY/C,K/D,K/56,13/.                          !
! MENT//D,K/48,30/.                              !
! MORE/C,E,K,N//13/:                             !
! MOTHER/C,K/D,K/16,13/.                         !
! MUCH/C,K//13,33/:                              !
! MUST/C,E,K,N//13,12/.                          !
! MYSELF/C,K//13,61,11/.                         !
! NAME/C,K/D,H,G,K/16,29/.                       !
! NECESSARY/C,K/D,K/29,17,9/:                    !
! NEITHER/C,K//29,17,10/:                        !
! NESS//D,K/48,14/.                              !
! NOT/C,E,K,N//29/.                              !
! OF/M,J,K/K/55/.                                !
! ONE/C,G,M/D,G,K/16,21/:                        !
! ONESELF/M,C,K//16,21,11/:                      !
! ONG//D,K/48,27/.                               !
! OU/M/M/51/:                                    !
! OUGHT/M,C,K/M,D,K/16,51/:                      !
! OUND//M,D,K/40,25/:                            !
! OUNT//M,D,K/40,30/:                            !
! OURSELVES/M,C,K//51,23,39,14/:                 !
! OUT/M,E,K,N//51/.                              !
! OW/K/K/42/.                                    !
! PAID/C,K/D,K/15,25/:                           !
! PART/C,K/D,K/16,15/.                           !
! PEOPLE/C,K//15/:                               !
! PERCEIVE/C,K//15,59,9,39/:                     !
! PERCEIVING/C,K//15,59,9,39,27/:                !
! PERHAPS/C,K//15,59,19/.                        !
! QUESTION/C,K//16,31/:                          !
! QUICK/C,K//31,5/:                              !
! QUITE/C,E,K,N//31/.                            !
! RATHER/C,K//23/.                               !
! RECEIVE/C,K//23,9,39/:                         !
! RECEIVING/C,K//23,9,39,27/:                    !
! REJOICE/C,K//23,26,9/:                         !
! REJOICING/C,K//23,26,9,27/.                    !
! RIGHT/C,K/D,K/16,23/.                          !
! SAID/C,K//14,25/.                              !
! SH/M/K/41/:                                    !
! SHALL/M,C,E,K,N//41/:                          !
! SHOULD/M,C,K//41,25/.                          !
! SION//D,K/40,29/.                              !
! SO/E,K,L//14/:                                 !
! SOME/L,C,K/D,K/16,14/.                         !
! SPIRIT/C,K/D,K/56,14/.                         !
! ST/M/K/12/:                                    !
! STILL/M,C,E,K,N//12/.                          !
! SUCH/C,K//14,33/.                              !
! TH/M/M/57/:                                    !
! THAT/M,C,E,K,N//30/:                           !
! THE/M,M,J,K/M,K/46/:                           !
! THEIR/M,C,K//56,46/:                           !
```

Table A.4.1 (continued)

```
! THEMSELVES/M,C,K//46,13,39,14/:          !
! THERE/M,C,K//16,46/:                     !
! THESE/M,C,E,K,N//24,46/:                 !
! THIS/M,C,K//57/:                         !
! THOSE/M,C,K//24,57/:                     !
! THROUGH/M,C,K/M,D,K/16,57/:              !
! THYSELF/M,C,K//57,61,11/.                !
! TIME/C,K/D,H,G,K/16,30/:                 !
! TION//D,K/48,29/.                        !
! TO/I,K,L//22/:                           !
! TODAY/L,C,K//30,25/:                     !
! TOGETHER/L,C,K//30,27,23/:               !
! TOMORROW/L,C,K//30,13/:                  !
! TONIGHT/L,C,K//30,29/.                   !
! UNDER/C,K/D,K/16,37/.                    !
! UPON/C,K/D,K/24,37/.                     !
! US/E,K,N//37/.                           !
! VERY/C,K//39/.                           !
! WAS/C,E,K,N//52/.                        !
! WERE/C,E,K,N//54/.                       !
! WH/M/M/49/:                              !
! WHERE/M,C,G,K/M,D,K/16,49/:              !
! WHICH/M,C,K//49/:                        !
! WHOSE/M,C,K//24,49/.                     !
! WILL/C,E,K,N//58/:                       !
! WITH/C,M,J,K/D,K/62/.                    !
! WORD/C,K/D,K/24,58/:                     !
! WORK/C,K/D,K/16,58/:                     !
! WORLD/C,K/D,K/56,58/:                    !
! WOULD/C,K//58,25/.                       !
! YOU/C,E,K,L//61/:                        !
! YOUNG/L,C,K//16,61/:                     !
! YOUR/L,E,K,L//61,23/:                    !
! YOURSELF/L,C,K//61,23,11/:               !
! YOURSELVES/L,C,K//61,23,39,14/.          !
```

Table A.4.1 (continued)

```
! 29,                                                                  !
! AESTHE/1,17,12,19,17///1,17,14,46/.                                  !
! ANEMONE/1,29,17,13,16,21///1,29,17,13,21,29,17/.                     !
! ATMOSPHERE/1,30,13,21,14,15,16,19///1,30,13,21,14,15,                !
!                                    19,57,17/.                        !
! BARONE/3,28,16,21///3,28,21,29,17/.                                  !
! BEGG/6,54///3,17,54/:                                                !
! BELL/6,7,7///3,17,7,7/:                                              !
! BELT/6,7,30///3,17,7,30/:                                            !
! BENEATH/3,34,2,57///6,29/:                                           !
! BEST/6,12///3,17,12/:                                                !
! BETT/6,30,30///3,17,30,30/.                                          !
! BLINDNESS/3,7,20,25,48,14///3,7,48,14/:                              !
! BLOSSOMED/3,7,21,14,16,14,25///3,7,21,14,14,21,13,43/.               !
! CENTIME/9,34,16,30///9,34,30,10,13,17/.                              !
! COLONE/9,21,7,16,21///9,21,7,21,29,17/:                              !
! CONCEIVE/22,18,9,17,10,39,17///22,18,9,39/:                          !
! CONCH/18,33///9,21,29,33/:                                           !
! CONG/9,48,27///18,27/.                                               !
! CREATION/9,23,2,48,29///9,23,17,32,29/.                              !
! DISC/50,9,0///25,10,14,9/.                                           !
! FORENAME/63,34,1,13,17///63,17,16,29/.                               !
! FRUITY/11,23,37,48,61///11,23,37,10,30,61/.                          !
! HERET/16,19,30///19,59,17,30/.                                       !
! LIONE/7,10,16,21///7,10,21,29,17/.                                   !
! MISH/13,10,41///13,10,14,19/.                                        !
! PERSEVER/15,59,14,16,17///15,59,14,17,39,59/.                        !
! PIONE/15,10,16,21///15,10,21,29,17/.                                 !
! PREAM/15,23,2,13///15,23,17,1,13/:                                   !
! PRED/15,23,43///15,23,17,25/.                                        !
! REAB/23,2,3///23,17,1,3/:                                            !
! REAC/23,2,9///23,17,1,9/:                                            !
! READJ/23,2,25,26///23,17,1,25,26/:                                   !
! READM/23,2,25,13///23,17,1,25,13/:                                   !
! REAFF/23,2,22///23,17,1,22/:                                         !
! REALLY/23,2,7,7,61///23,17,32,61/:                                   !
! REAP/23,2,15///23,17,1,15/:                                          !
! REARR/23,2,23,23///23,17,28,23/:                                     !
! REASS/23,2,14,14///23,17,1,14,14/:                                   !
! REDE/23,43,17///23,17,25,17/:                                        !
! REDI/23,43,10///23,17,25,10/:                                        !
! RENAME/23,34,1,13,17///23,17,16,29/:                                 !
! RENOU/23,34,51///23,17,29,51/:                                       !
! REVER/23,16,17///23,17,39,59/.                                       !
! SEVERE/14,16,17,17///14,17,39,59,17/.                                !
! SHOULDER/41,25,59///41,51,7,25,59/.                                  !
! SPHERE/14,15,16,19///14,15,19,59,17/.                                !
! SWORD/14,24,58///14,58,21,23,25/.                                    !
! THENC/46,29,9///57,48/.                                              !
! UNDERIVED/16,37,10,39,43///37,29,25,59,10,39,43/.                    !
! WAS/52,52///3,61,0,52/.                                              !
! WHEREVER/16,49,39,59///49,59,16,17/.                                 !
```

Dictionary of Exceptions

Table A.4.2

Appendix 5

SPEEDING-UP TECHNIQUES

It is well known that whenever it is necessary to access dictionaries frequently, any process slows down. Thus the best way to optimise the use of dictionaries is to guarantee that whenever an access is going to be made, it is really necessary. One way of doing this is by filtering out as many cases as possible that will result in unsuccessful searches.

With this in mind for the translation from English into Braille Grade II, speeding-up techniques have been introduced into the process to improve translation times, namely with the use of positional n-grams to guarantee that tree following is only done when it is really required.

This technique, developed by Riseman and Ehrich (Riseman and Ehrich, 1971), is an effective means of extracting large amounts of the information from the dictionary in a readily retrievable form, at a relatively modest cost of storage. This information is in the form of a database of quantised n-grams (Riseman and Hanson, 1974). These n-grams, for any n, may be either positional or non-positional, depending on the amount of information stored and the method by which the information is extracted from the dictionary. For example, corresponding to the non-positional letter pair

probabilities, there is a 27 by 27 binary matrix whose entries are 1 if and only if that corresponding letter pair appears in some word in the dictionary. The rest of the entries will have a zero value. It is worth noting that spaces must be considered in non-positional information, requiring 27 characters; positional n-grams using 26 characters, automatically include this information, since a space always precedes the first position and follows the last position. Positional digram matrices are constructed to take into consideration the relative positions of the letters within words. One binary matrix exists for each distinct pair of positions; for example, for 6 letter words there are 15 positional digrams required to contain all pairwise positional information contained in the dictionary. These figures are obtained according to the following formula:

NPD = (NL)! / ((NL-2)! * 2!) , where

NPD is the number of positional digrams,

NL is the number of letters in words under consideration,

and ! represents factorial.

This formula emerges from the different possible permutations of 6 elements arranged pairwise.

The set of all positional digrams allows one to obtain the same information that would be obtained from an associative

memory that could only be asked the following type of question: Is there some word in the dictionary that has the letters 1 and 2 in positions i and j, respectively? Letters 1 and 2 may be any letter from 'a' to 'z'.

From this, the extension to binary trigrams is quite straightforward. In fact, a dictionary of words of length 1 can be viewed as a binary 1-gram organised in a different way. Since very few of 26**1 possible 1-letter words actually exist in the dictionary, rather than construct an extremely large and sparse matrix, the dictionary itself is a list of the nonzero entries.

Here, for English into Braille Grade II translation, positional binary digrams are used to minimise tree following. Binary digrams are used both for the abbreviations and contractions dictionary and for the exceptions dictionary. For the first dictionary, two positional binary digrams are constructed, both of them considering only the first two letters of the Braille contractions and abbreviations in the dictionary. The purpose of this is to be able to detect with a binary test whether a certain pair of characters from the input stream is valid or not as a Braille contraction or abbreviation.

The first of these digrams was built according to the Braille contractions and abbreviations that can be used at the beginning of the word, and the second, according to the Braille contractions and abbreviations that can be used

anywhere, except at the beginning of the word. This, of course, involves the redundancy generated by repeating the Braille contractions and abbreviations that can be used anywhere in the word; however, the splitting was considered convenient because in both digrams the density is reduced considerably, implying time saving mainly in tree following, which is the most time-consuming part of the translation process.

At this point it may be convenient to state some statistics regarding digrams for the English Language and for Braille contractions and abbreviations. The digram for English Language has a density that approaches 60% (Riseman and Ehrich, 1979). The digram for Braille contractions and abbreviations that may occur at the beginning of the word has a density of 11.7% (figure A.5.1), and the digram for Braille contractions and abbreviations that occur not at the beginning of the word has a density of 6.5% (figure A.5.2). It is important to note that according to the average length of the English word (5.8 characters (Haynes and Siems, 1979)), 17.2% of the matching process will be carried out using the first digram and the remaining 82.8% using the second digram, which has a very low density and will avoid most of the useless tree following.

In both digrams (figures A.5.1 and A.5.2), a binary 1 at an intersection indicates that there are Braille contractions or abbreviations starting with the corresponding two-letter combination. the first character corresponds to the row of

the intersection; the second character corresponds to the column.

For the exceptions dictionary only one digram is used, considering the first two letters of exceptions, so that it is possible to guarantee that when tree following is performed, there is a high possibility of success in finding an exception. The positional digram for exceptions is shown in figure A.4.3, and has a density of 4.4%. However, it is worth noting that this figure is entirely dependent on the number of exceptions included in the dictionary, and would vary according to this (or more precisely, according to the number of different two initial letters in the exceptions dictionary).

FIRST LETTER	SECOND LETTER A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	A		1	1			1	1				1		1				1	1							
	B				1							1						1			1				1	
	C	1						1							1											
	D	1			1										1											
	E			1					1					1				1				1				
	F	1							1						1			1								
	G							1							1			1								
	H	1			1				1																	
	I												1	1						1						
	J																				1					
	K													1												
	L				1				1						1											
	M	1													1						1				1	
	N	1			1										1											
	O					1								1							1		1			
	P	1			1																					
	Q																				1					
	R	1			1				1																	
	S	1						1							1	1				1	1					
	T							1	1						1											
	U													1		1			1							
	V				1																					
	W	1			1			1	1						1											
	X																									
	Y														1											
	Z																									

Density of the digram: 11.7%

Binary Digram for Braille Contractions and Abbreviations
at the Beginning of the Word
(Letter positions 1 and 2)

Figure A.5.1

FIRST LETTER	SECOND LETTER	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	A												1		1				1		1						
	B		1										1														
	C			1					1																		
	D	1			1					1																	
	E	1			1										1				1				1				
	F						1									1						1					
F	G							1	1																		
I	H																										
R	I														1						1						
S	J																										
T	K														1												
	L					1																					
L	M	1				1																					
E	N					1																					
T	O						1								1							1		1			
T	P	1																									
E	Q																										
R	R									1																	
	S								1	1						1					1						
	T								1	1																	
	U														1												
	V																										
	W								1	1						1											
	X																										
	Y																										
	Z																										

Density of the digram: 6.5%

Binary Digram for Braille Contractions and Abbreviations
Not at the Beginning of the Word
(Letter positions 1 and 2)

Figure A.5.2

FIRST LETTER	SECOND LETTER	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	A					1									1						1						
	B	1				1							1														
	C					1										1			1								
	D					1				1																	
	E															1											
	F																		1								
F	G																										
I	H					1																					
R	I																										
S	J																										
T	K																										
	L									1																	
L	M									1																	
E	N																										
T	O																										
T	P	1				1				1									1								
E	Q																										
R	R					1																					
	S					1			1								1							1			
	T								1																		
	U														1												
	V																										
	W	1							1																		
	X																										
	Y																										
	Z																										

Density of the digram: 4.4%

Positional Binary Digram for EBT Exceptions
(Letter positions 1 and 2)

Figure A.5.3

Appendix 6

PROGRAM EXECUTION AND LISTING

1. Running the Translation Program.

The instructions for executing the program will vary depending on the machine and operating system used. Specifically, to run the program on an Apple microcomputer, first it is recommended to set the file prefix to "EBT:". Then the "X" command should be used. The response to the execution-file query should be the name of the file which contains the program code. At this moment, the translation program is running, and the names for the input and output files are requested. If the input file does not exist, the program terminates and the system is re-initialised. Afterwards, the user is requested to decide whether or not Grade II translation is required. If this is the case, the dictionaries of abbreviations, contractions and exceptions are loaded, and then the translation process begins. If Grade II Braille is not desired, Grade I translation starts immediately.

In relation to the input text submitted to the translator there are some control characters which will be required for translating English into Braille adequately. These are listed next.

- Italics sign. The symbol for specifying that a word should be italicised is "#". This symbol must precede the italicised word.

- Bypass contraction. For specifying that a certain string should not be contracted, an "@" (at sign) should precede it. This sign may appear anywhere within a word, and from this point on, the following text will be translated as Grade I. The bypassing of contractions is terminated either by a blank or by another "@", thus enabling contraction of just part of a word.

- Letter sign. An ampersand ("&") at the beginning of a string specifies that the following characters are to be interpreted as individual letters rather than as a word.

- New paragraph. A backslash ("\") in the input text specifies provision of a new paragraph in the corresponding Braille output.

If letter, bypass and/or italics symbols must be specified together, the precedence used should be: first, the bypass (no-contraction) character, then the letter symbol, and finally the italics symbol.

2. The Program Listing.

The listing of the program for English into Braille Grade II translation is provided next. It is internally commented for ease of understanding.

The individual parts of the program are referred to and described in chapter 5, section 5.2.

```
program ebt;

  const
    nchars = 26;
    maxnstacks = 10;
    wordsize = 20;
    linesize = 26;
    linesperpage = 16;
    numbersymbol = 60;
    lettersymbol = 48;
    signsymbol = 48;
    closequote = 52;
    letter = '&';
    bypasschar = '@';
    paragraph = '\';
    italics = '#';

  type
    asciirange = 32..95;
    delimiter = ' '..'?';
    number = '0'..'9';
    bchset = 0..63;

    translref = ^transls; (* pointer to translations *)
    transls = record
                  transla          : integer;
                  ntransla         : translref;
                end;
    transiref = ^transitions;  (* pointer to transitions *)
    transitions = record
                   state           : char;
                   nexttransi      : transiref;
                 end;
    contref = ^conts;   (* pointer to contractions *)
    conts = record
                  ichars           : transiref;
                  braillerep       : translref;
                  transibw         : transiref;
                  transinbw        : transiref;
                  transl           : translref;
                  nextcontraction  : contref;
                end;
    noderef = ^node;           (* pointer to node *)
    node = record
             key                   : integer;
             contra                : contref;
             left,right            : noderef;
           end;
    stackref = ^stack;         (* pointer to stack *)
    stack = record
              level                : integer;
              chs                  : char;
              brreps               : integer;
              posatbw              : boolean;
              depth                : integer;
              presstate            : char;
              acnode               : noderef;
              accont               : contref;
              acichars             : transiref;
```

```
        numichars                    : integer;
        acbraillerep                 : translref;
        actransi                     : transiref;
        actransl                     : translref;
        next                         : stackref;
      end;
var
  inp,out,inp2 : text;
  capital,lastcharblank,lastchdot,
  wordcomplete,linecomplete,wordfitinline,
  numflag,letterflag,exceptions,quotation,
  gradeii,bypasscontraction,newparagraph,
  bypassgradeii,positionatbw,posblankcont : boolean;
  digrambw,digramnbw,excepdigram : array[1..nchars,1..nchars]
                                   of boolean;
  brrep,auxbrrep,xx,nletters,
  linenumber,linecount,wlength,stacklevel : integer;
  delims                                  : set of delimiter;
  numbers                                 : set of number;
  ordbchset                               : set of bchset;
  inputch,presentstate                    : char;

  sref                                    : stackref;
  stacks       : array[1..maxnstacks] of stackref;
  conthead                                : contref;
  hbwtransi,hnbwtransi,icharshead         : transiref;
  brarephead,translhead                   : translref;
  root,exceproot                          : noderef;
  braillechrep    : array [asciirange] of integer;
  word            : array [1..wordsize] of integer;
  line            : array [1..linesize] of integer;
  heap                                    : ^integer;
```

```
procedure openfiles;
 var
   inpfile,outfile : string;
 begin  (* openfiles *)
   write('Input file name:  ');
   readln(inpfile);
   reset(inp,inpfile);
   write('Output file name: ');
   readln(outfile);
   rewrite(out,outfile);
   reset(inp2,'abcont.text');
 end  (* openfiles *);

 procedure initialise;
 var
   answer          : char;
   i,j             : integer;
   newloc          : stackref;

 (* initialisation of global variables *)
 begin  (* initialise *)
   wordcomplete := false;
   linecomplete := false;
   wordfitinline := true;
   numflag := false;
   letterflag := false;
   exceptions := false;
   quotation := false;
   gradeii := false;
   bypasscontraction := false;
   newparagraph := false;
   bypassgradeii := false;
   linenumber := 0;
   linecount := 0;
   wlength := 0;
   stacklevel := 0;

   delims := [' ','"','#','$','%','&','@','\','~','(',')','*','+'
              ',','_','.','/',':',';','<','=','>','?','!','[',']']
   numbers := ['0','1','2','3','4','5','6','7','8','9'];

   write('Do you want Grade II translation? ');
   readln(answer);
   if (answer = 'Y') or (answer = 'y') then
     begin
     gradeii := true;
     positionatbw := true;
     posblankcont := false;
     for i := 1 to nchars do
       for j := 1 to nchars do
         begin
         digrambw[i,j] := false;
         digramnbw[i,j] := false;
         excepdigram[i,j] := false;
         end;
     new(newloc);
     sref := newloc;
     conthead := nil;
     icharshead := nil;
```

```
    brarephead := nil;
    hbwtransi := nil;
    hnbwtransi := nil;
    translhead := nil;
    for i := 1 to maxnstacks do
      stacks[i] := nil;
  end
end (* initialise *);

procedure mapbrailletoascii;
(* procedure in  charge of mapping the braille characters
  to the ASCII character set *)
var
  i : integer;
begin (* mapbrailletoascii *)
  braillechrep[ord('A')] := 1;
  braillechrep[ord('B')] := 3;
  braillechrep[ord('C')] := 9;
  braillechrep[ord('D')] := 25;
  braillechrep[ord('E')] := 17;
  braillechrep[ord('F')] := 11;
  braillechrep[ord('G')] := 27;
  braillechrep[ord('H')] := 19;
  braillechrep[ord('I')] := 10;
  braillechrep[ord('J')] := 26;
  braillechrep[ord('K')] := 5;
  braillechrep[ord('L')] := 7;
  braillechrep[ord('M')] := 13;
  braillechrep[ord('N')] := 29;
  braillechrep[ord('O')] := 21;
  braillechrep[ord('P')] := 15;
  braillechrep[ord('Q')] := 31;
  braillechrep[ord('R')] := 23;
  braillechrep[ord('S')] := 14;
  braillechrep[ord('T')] := 30;
  braillechrep[ord('U')] := 37;
  braillechrep[ord('V')] := 39;
  braillechrep[ord('W')] := 58;
  braillechrep[ord('X')] := 45;
  braillechrep[ord('Y')] := 61;
  braillechrep[ord('Z')] := 53;
  ordbchset := [1,3,9,25,17,11,27,19,10,26,5,7,13,29,21,15,31,
                23,14,30,37,39,58,45,61,53];
  (* ordbchset is the ordinary braille character set *)
  for i := ord('1') to ord('9') do
    braillechrep[i] := braillechrep[i+16];
  braillechrep[ord('0')] := 26;
  braillechrep[ord(' ')] := 0;
  braillechrep[ord('!')] := 22;
  braillechrep[ord('?')] := 38;
  braillechrep[ord('"')] := 38;
  braillechrep[ord('''')] := 4;
  braillechrep[ord('(')] := 54;
  braillechrep[ord(')')] := 54;
  braillechrep[ord(',')] := 2;
  braillechrep[ord('-')] := 36;
  braillechrep[ord('.')] := 50;
  braillechrep[ord(':')] := 18;
```

```
  braillechrep[ord(';')] := 6;
  braillechrep[ord('/')] := 12;
  braillechrep[ord('+')] := 22;
  braillechrep[ord('=')] := 54;
  braillechrep[ord('$')] := 50;
  braillechrep[ord('%')] := 18;
  braillechrep[ord('*')] := 20;
  braillechrep[ord('[')] := 32;
  braillechrep[ord(']')] := 54;
  braillechrep[ord('#')] := 40; (* char for specifying italics *)
  braillechrep[ord('&')] := 48; (* char for letter sign *)
  end (* mapbrailletoascii *);

procedure buildabconttransnetw;
(* Procedure in charge of building up the dictionary
     required for the translation process. Specifically,
     it builds the abbreviation and contraction
     transition network.                                  *)
var
  n : integer;
function tree (n:integer) : noderef;
(* Construction of a perfectly balanced tree with n nodes.
   Recursive function                                     *)
    var
      newnode : noderef;
      x,nl,nr : integer;
      ch      : char;
    function formcontraction (var ch : char) : contref;
    (* Insertion of contractions in the dictionary. Recursive
       procedure that calls itself as many times as contractions
       with the same two initial characters exist.    *)
      var
        newcont : contref;
        ch1,ch2 : char;
      function ftransllist : translref;
      (* Formation of translation list                  *)
        var
          newtransl : translref;
          num       : integer;
        begin  (* ftransllist *)
          read(inp2,num);
          read(inp2,ch);
          read(inp2,ch);
          new(newtransl);
          newtransl^.transla := num;
          if ch = '/' then
            newtransl^.ntransla := nil
          else
            newtransl^.ntransla := ftransllist;
          ftransllist := newtransl;
        end  (* ftransllist *);

      function ftransilist : transiref;
      (* Insertion of input characters into dictionary
         and of lists of transitions.                 *)
        var
          newtransi : transiref;
        begin  (* ftransilist *)
```

```
      read(inp2,ch);
      if ch = '/' then
        ftransilist := nil
      else
        begin
        if ch = ',' then
          read(inp2,ch);
        new(newtransi);
        newtransi^.state := ch;
        newtransi^.nexttransi := ftransilist;
        ftransilist := newtransi;
        end
    end  (* ftransilist *);

  begin  (* formcontraction *)
  if ch <> '.' then
    begin
      icharshead := ftransilist;
      ch1 := icharshead^.state;
      ch2 := icharshead^.nexttransi^.state;
      x := nchars * (ord(ch1) - ord('A')) +
           (ord(ch2) - ord('A')) + 1;
      if exceptions then
        brarephead := ftransllist;
      hbwtransi := ftransilist;
      if hbwtransi <> nil then
        digrambw[ord(ch1) - ord('A') + 1,
                 ord(ch2) - ord('A') + 1] := true;
      hnbwtransi := ftransilist;
      if hnbwtransi <> nil then
        digramnbw[ord(ch1) - ord('A') + 1,
                  ord(ch2) - ord('A') + 1] := true;
      if exceptions then
        excepdigram[ord(ch1) - ord('A') + 1,
                    ord(ch2) - ord('A') + 1] := true;
      translhead := ftransllist;
      readln(inp2,ch);
      new(newcont);
      with newcont^ do
        begin
        ichars := icharshead;
        if exceptions then
          braillerep := brarephead;
        transibw := hbwtransi;
        transinbw := hnbwtransi;
        transl := translhead;
        nextcontraction := formcontraction(ch)
        end;
      formcontraction := newcont
    end  (* if then *)
  else
    formcontraction := nil
  end  (* formcontraction *);

begin  (* tree *)
if n = 0 then
  tree := nil
else
```

```
    begin
    conthead := formcontraction(ch);
    ch := ' ';
    nl := n div 2;
    nr := n - nl - 1;
    new(newnode);
    with newnode^ do
      begin
      key := x;
      contra := conthead;
      left := tree(nl);
      right := tree(nr)
      end;
    tree := newnode
    end
  end  (* tree *);

begin  (* buildabconttransnetw *)
  readln(inp2,n);
  root := tree(n);
  exceptions := true;
  readln(inp2,n);
  exceproot := tree(n);
  exceptions := false;
end  (* buildabconttransnetw *);

procedure getbraillerep;
(* Procedure for reading the next input character and getting its
   corresponding Braille representation *)
const
  deface = 32;  (* for treating lowercase as uppercase *)
begin (* getbraillerep *)
repeat
  inputch := inp^;
  get(inp);
  if eoln(inp) and not eof(inp) then
    get(inp);
until (not lastcharblank) or (inputch <> ' ') or eof(inp);
if ord(inputch) > 96 then
  begin
  inputch := chr(ord(inputch) - deface);
  capital := true
  end;
newparagraph := (inputch = paragraph);
bypassgradeii := (inputch = bypasschar);
brrep := braillechrep[ord(inputch)];
if (lastcharblank or lastchdot) and (inputch = '.') then
  brrep := 4;           (* representation of ellipsis *)
lastcharblank := (inputch = ' ');
lastchdot := (inputch = '.');
if inputch = '"' then
  begin
  if quotation then
    begin
    brrep := closequote;
    quotation := false
    end
  else
```

```
      quotation := true
    end;
  if inputch in ['*','[',']','%'] then
    case inputch of
      '*' : auxbrrep := 20;
      '[' : auxbrrep := 54;
      ']' : auxbrrep := 4;
      '%' : auxbrrep := 15
    end;
  if (inputch = bypasschar) or (inputch = letter) then
    bypasscontraction := not bypasscontraction
  else
    if bypasscontraction then
      if inputch = ' ' then
        bypasscontraction := false;
  end  (* getbraillerep *);

  function locatekey(x : integer; t : noderef) : noderef;
  (* function for locating existing key in dictionary *)
  var
    found,endofs : boolean;
  begin
  found := false;
  endofs := false;
  while (t <> nil) and (not found) and (not endofs) do
    begin
    if t^.key = x then
      found := true
    else
      if t^.key > x then
        endofs := true
      else
        begin
        if t^.right <> nil then
          begin
          if t^.right^.key > x then
            t := t^.left
          else
            t := t^.right
          end
        else
          t := t^.left
        end
    end;
  if found then
    locatekey := t
  else
    locatekey := nil
  end  (* locatekey *);
```

```
procedure formbraword;
 (* Formation of a Braille grade I or grade II word *)
 var
  stackpos        : integer;
  topofstack      : stackref;
  contatoutlevel  : boolean;
  numtrans        : integer;

 procedure push(var topofstack : stackref);
 (* this procedure creates a new location on top of a stack *)
   var
     newloc : stackref;
   begin (* push *)
   new(newloc);
   newloc^.next := topofstack;
   topofstack := newloc
   end (* push *);

 procedure pop(var topofstack : stackref);
 (* this procedure eliminates the top element from a stack *)
   var
     temp : stackref;
   begin (* pop *)
   if topofstack <> nil then
     begin
     temp := topofstack;
     topofstack := topofstack^.next;
  (* dispose(temp) *) (* since 'dispose' is not implemented in
                         apple pascal, 'mark' and 'release'
                         are used in the outer block *)
     end
   else
     writeln('Error: Popping element from empty stack.')
   end (* pop *);

 procedure checkfornums;
 (* number translation *)
   begin (* checkfornums *)
   if inputch in numbers then
     begin
     if (not numflag) and (stacklevel = 0) then
       begin
       wlength := succ(wlength);
       word[wlength] := numbersymbol;
       numflag := true;
       letterflag := false
       end
     end
   else
     begin
     if numflag or letterflag then
       begin
       if inputch in ['A'..'Z'] then
         begin
         if (not letterflag) and not
            ((inputch = 'T') or (inputch = 'R') or
             (inputch = 'S') or (inputch = 'N')) then
           begin
           wlength := succ(wlength);
```

```
        word[wlength] := lettersymbol;
          letterflag := true
          end
      end
    else
      begin
      if inputch in ['+','-','/','*','='] then
        begin
        wlength := succ(wlength);
        word[wlength] := 0;
        wlength := succ(wlength);
        word[wlength] := signsymbol;
        if inputch = '/' then
          brrep := 50
        else
          if inputch = '*' then
            brrep := 38;
        end;
        letterflag := false
        end;
      numflag := false
      end
    end
  end (* checkfornums *);

procedure translate;
  (* Procedure for coordinating the translation
     of the input text *)
  var
    possiblecont,brrepsinordbchset : boolean;
    blankgen,continue              : boolean;
    i,x,decr                       : integer;

  function getkey(topofstack:stackref; inputch:char) : integer;
  (* This function computes the key value, according to the
     input character and the one on top of the stack *)
  var
    chl : char;
  begin (* getkey *)
  chl := topofstack^.chs;
  getkey := nchars * (ord(chl)-ord('A')) +
            (ord(inputch)-ord('A')) + 1
  end (* getkey *);

  function checkdigram(topofstack : stackref;
                       inputch : char) : boolean;
  (* This function checks for the possibility of contracting
     any two contiguous characters. Binary digrams are used
     for this purpose. *)
  begin
  with topofstack^ do
    begin
    if posatbw then (* digram for contractions at the
                       beginning of the word *)
      checkdigram := digrambw[ord(chs) - ord('A') + 1,
                              ord(inputch) - ord('A') + 1]
    else (* digram for contractions not at the beginning
            of the word *)
```

```
    checkdigram := digramnbw[ord(chs) - ord('A') + 1,
                             ord(inputch) - ord('A') + 1]
  end
end (* checkdigram *);

procedure getnextcont;
(* This procedure gets the next contraction in the
   dictionary, if there is one with the same two
   starting characters *)
begin (* getnextcont *)
with sref^ do
  begin
  continue := false;
  while not continue and (accont^.nextcontraction <> nil) do
    begin
    accont := accont^.nextcontraction;
    acichars := accont^.ichars;
    if posatbw then
      actransi := accont^.transibw
    else
      actransi := accont^.transinbw;
    actransl := accont^.transl;
    if actransi <> nil then
      continue := true
    end
  end
end (* getnextcont *);

procedure getposatnextcont;
(* This procedure gets the current position in the word
   being analised, in the following word (if there is one
   with the same 2 starting chars) *)
var
  oldichars,cmpichars : transiref;
  posfound            : boolean;
  i                   : integer;
begin (* getposatnextcont *)
posfound := false;
continue := true;
oldichars := sref^.accont^.ichars;
while continue and not posfound do
  begin
  i := 1;
  cmpichars := oldichars;
  getnextcont;
  with sref^ do
    begin
    if continue then
      begin
      while (i <= numichars) and continue do
        begin
        if acichars <> nil then
          begin
          if cmpichars <> nil then
            begin
            if acichars^.state <> cmpichars^.state then
              continue := false;
            acichars := acichars^.nexttransi;
```

```
            cmpichars := cmpichars^.nexttransi
            end
          else
            continue := false
          end
        else
          begin
          if cmpichars <> nil then
            continue := false
          end;
        i := succ(i);
        if presentstate = ' ' then
          presentstate := actransi^.state;
        if (actransi^.state <> presentstate) and
          (actransi^.nexttransi <> nil) then
          actransi := actransi^.nexttransi
        end;
      if (acichars^.state = inputch) and
        (i = numichars+1) and continue then
        posfound := true;
      end (* if continue *)
    end (* with *)
  end (* while *);
if not posfound then
  presentstate := 'N'
end (* getposatnextcont *);

procedure gettransition;
(* procedure for performing a state transition according
  to the information in the dictionary *)
var
  cond : boolean;
  x    : integer;

procedure getnextstate;
begin
sref^.actransi := sref^.actransi^.nexttransi;
presentstate := sref^.actransi^.state
end;

procedure cdtransition;
(* transition from state 'C' and 'D' *)
begin (* cdtransition *)
if inputch <> sref^.acichars^.state then
  getposatnextcont;
if presentstate <> 'N' then
  if sref^.acichars^.nexttransi = nil then
    getnextstate;
end (* cdtransition *);

procedure etransition;
(* transition from state 'E' *)
var
  y : integer;
begin (* etransition *)
y := sref^.acnode^.key;
if inputch = ' ' then
  presentstate := sref^.actransi^.nexttransi^.state
```

```
else
  if inputch in delims then
    if (y=118) or (y=573) or (y=577) or
       ((y=191) and (sref^.next^.brreps =14)) then
      presentstate := 'N'     (* do not allow 'was', 'were', 'his',
                                 'enough' before punctuation marks *)
    else
      presentstate := sref^.actransi^.nexttransi^.state
  else
    begin
    presentstate := sref^.actransi^.nexttransi^.nexttransi^.state;
    if presentstate = 'L' then
      getposatnextcont;
    if presentstate = 'L' then
      getnextstate;
    end
end (* etransition *);

procedure ftransition;
(* transition from state 'F': consideration of 'COM', 'CON', 'DIS' *)
var
  cond1,cond2,cond3,cond4 : boolean;
begin (* ftransition *)
nletters := 3;  (* number of letters in 'COM', 'CON', 'DIS' *)
x := sref^.acnode^.key;
decr := sref^.numichars - nletters + 1;
if sref^.posatbw then
  if inputch in delims then
    presentstate := 'N'
  else
    begin
    cond1 := (x=87) and (inputch='T');  (* 'DIS' before 'T' *)
    cond2 := (x=67) and (stacks[1]^.chs='N') and ((inputch='E') or
             (inputch = 'G') or (inputch = 'K'));  (* 'CON' before 'E,G,K
    cond3 := (x=87) and (inputch='H');  (* 'DIS' before 'H' *)
    if cond1 then
      begin
      push(stacks[1]);
      sref^.next := stacks[1]^.next;
      stacks[1]^ := sref^;
      with stacks[1]^ do
        begin
        depth := succ(depth);
        numichars := pred(numichars);
        brreps := 14; (* mapping for 'S' is 14, instead of m
                         mapping stored for 'ST' *)
        end;
      contatoutlevel := false;
      sref^.chs := inputch;
      sref^.brreps := 30        (* mapping for 'T' *)
      end
    else
```

```
      if cond2 or cond3 then  (* 'CON' and 'DIS' will not be
                                 contracted in these cases *)
        presentstate := 'N'
      else
      getnextstate;
    end
  else
    presentstate := 'N'
end (* ftransition *);

procedure gtransition;
(* procedure for getting the transition from
   state 'G' (special cases) *)
var
  cond1,cond2 : boolean;
begin (* gtransition *);
x := sref^.acnode^.key;
cond1 := (((x=118) or (x=378)) and
         ((inputch='D') or (inputch='N') or (inputch='R')))
         (* 'ENCE' or 'ONE' followed by 'D', 'N', or 'R' *)
         or
         ((x=31) and ((inputch='A') or (inputch='D') or
                      (inputch='E') or (inputch='N') or
                      (inputch='R')))
         (* 'BE' followed by 'A', 'D', 'E', 'N' or 'R' *)
         or
         ((x=187) and ((inputch='D') or (inputch='N') or
          (inputch='R')));
                 (* 'HERE' followed by 'D', 'N' or 'R' *)
cond2 := ((x=28) and (inputch='E'))
                 (* 'BB' followed by 'BLE' *)
         or
         ((x=82) and (inputch='Y'))
                (* 'DD' followed by 'DAY' *)
         or
         ((x=136) and (inputch='R'))
               (* 'FF' followed by 'FOR' *)
         or
         ((x=339) and (inputch='T'))
               (* 'NAME' followed by 'MENT' *)
         or
         ((x=503) and (inputch='T'));
               (* 'TIME' followed by 'MENT' *)
if x in [28,82,136,339,503] then
  if inputch in delims then
    decr := 1
  else
    decr := 2
else
  if inputch in delims then
    decr := 0
  else
    decr := 1;
if cond1 or cond2 then
  presentstate := 'N'
else
  getnextstate;
if (presentstate='M') and (inputch in delims) then
```

```
  presentstate := 'K';
end (* gtransition *);

procedure htransition;
(* procedure for getting transitions in the middle
   of the word *)
begin (* htransition *)
x := sref^.acnode^.key;
if inputch in delims then
  if (x=339) or (x=503) then
    presentstate := 'K'
  else
    presentstate := 'N'
else
  begin
  decr := 1;   (* delayed decision with deface = 1 *)
  getnextstate;
  if ((x=28) and (inputch <> 'L')) or
                    (* check for 'BB' followed by 'BLE' *)
     ((x=82) and (inputch <> 'A')) or
                    (* 'DD' followed by 'DAY' *)
     ((x=136) and (inputch <> 'O')) or
                    (* 'FF' followed by 'FOR' *)
     ((x=339) and (inputch <> 'N')) or
                    (* 'NAME' followed by 'MENT' *)
     ((x=503) and (inputch <> 'N')) then
                    (* 'TIME' followed by 'MENT' *)
    getnextstate;
  end
end (* htransition *);

procedure itransition;
(* procedure for getting transitions for words whose
  contraction depends on their position within the line,
  in relation to the word that follows *)
begin (* itransition *)
if posblankcont then
  begin
  if (sref^.actransl^.transla = 22) or
     (sref^.actransl^.transla = 52) then
    nletters := 2;          (* # of letters in 'TO' or 'BY' *)
  decr := sref^.depth - nletters;
  if (inputch in delims) or (inputch in numbers) then
    if brrep <> 4 then
      begin
      getnextstate;
      posblankcont := false
      end
    else
      begin
      presentstate := 'N';
      blankgen := true;
      posblankcont := false
      end
  else
    begin
    if (linecount + sref^.depth) > linesize then
      if (linecount + nletters) < linesize then
```

```
        begin (* the first word fits in the line *)
        presentstate := 'N';
        blankgen := true;
        posblankcont := false
        end
    end
  end
else
  begin
  decr := 0;
  if inputch = ' ' then
    begin
    posblankcont := true;
    nletters := sref^.depth
    end
  else
    if (inputch in delims) or (inputch in numbers) then
      presentstate := 'N'
    else
      presentstate :=
            sref^.actransi^.nexttransi^.nexttransi^.state
  end;
if presentstate = 'L' then
  getposatnextcont;
if presentstate = 'L' then
  getnextstate;
end (* itransition *);

procedure jtransition;
(* procedure for getting transitions when
   contracting spaces *)
var
  t : integer;
begin (* jtransition *)
if posblankcont then
  begin
  if not (inputch in delims) and
        not (inputch in numbers) then
    begin
    decr := sref^.depth - nletters;
    if stacks[stackpos + 1] <> nil then
     begin
     if stacks[stackpos+1]^.actransl <> nil then
      begin
      if stacks[stackpos+1]^.actransl^.ntransla = nil then
        begin
        t := stacks[stackpos+1]^.actransl^.transla;
        if (t<>47) and (t<>63) and (t<>55) and (t<>46) and
        (t<>57) and (t<>62) and (t<>58) then
          begin
          presentstate := 'N';
          blankgen := true
          end
        else
          if ((t=58) and (decr=2) and (inputch <> 'T')) or
             ((t=47) and (decr=2) and (inputch <> 'D')) then
            begin
            presentstate := 'N';
```

```
            blankgen := true
            end
        end
      else
        begin
        presentstate := 'N';
        blankgen := true
        end
      end
    else
      begin
      presentstate := 'N';
      blankgen := true
      end
    end
  else
    begin
    presentstate := 'N';
    blankgen := true
    end;
  if (presentstate = 'N') and (stackpos = 1) then
    posblankcont := false
  end
else
  if (sref^.depth > (nletters + 1)) or
  ((sref^.depth = nletters + 1) and
    (inputch <> ' ')) then
  begin
  decr := sref^.depth - nletters;
  blankgen := true;
  presentstate := 'N'
  end
  else
  begin
  decr := 0;
  nletters := sref^.depth;
  if stackpos = 1 then
    posblankcont := false;
  if inputch <> ' ' then
    getnextstate
  end
end
else
  begin
  decr := 0;
  nletters := sref^.depth;
  if inputch in ['A','F','O','T','W'] then
    begin
    if stackpos = 1 then
      posblankcont := true
    end
  else
    begin
    presentstate := 'N';
    blankgen := true
    end
  end
end (* jtransition *);
```

```
procedure mtransition;
(* procedure for getting transitions for words which
   will be contracted but which may form part of a
   longer contraction *)
var
  cond,posfound : boolean;
begin
cond := false;
if eof(inp) then
  presentstate := 'N'
else
  begin
  continue := true;
  if sref^.actransi^.nexttransi <> nil then
    begin
    if (sref^.actransi^.nexttransi^.state = 'J') and
       (inputch <> ' ') then
      begin
      decr := 0;
      cond := true
      end
    end
  else
    cond := true;
  if cond then
    begin
    if ((sref^.acichars = nil) or
        (sref^.acichars^.state <> inputch)) and
       continue then
      getposatnextcont
    end;
  if continue then
    begin
    getnextstate;
    if presentstate = 'J' then
      begin
      decr := 0;
      if inputch = ' ' then
        begin
        if (stackpos = 2) and
           (stacks[1]^.presstate = 'I') then
          presentstate := 'N'
        else
          begin
          if stackpos = 1 then
            posblankcont := false;
          stacks[stackpos]^.presstate := 'J';
          stacks[stackpos]^.actransi :=
              stacks[stackpos]^.actransi^.nexttransi
          end
        end
      else
        begin
        if sref^.depth > 1 then
          blankgen := true;
        presentstate := 'N'
        end
```

```
      end
    end
  else
    presentstate := 'N'
  end
end (* mtransition *);
begin (* gettransition *)
  case presentstate of
    ' ' : begin
          if sref^.actransi = nil then
            getposatnextcont
          else
            presentstate := sref^.actransi^.state;
          if (stackpos > 1) and
            (presentstate = 'I') then
            if stacks[stackpos-1]^.presstate = 'I' then
              begin
              getposatnextcont;
              if presentstate = ' ' then
                presentstate := sref^.actransi^.state
              end;
          if stackpos > 1 then
            if (stacks[stackpos-1]^.actransl^.transla = 23) and
              (stacks[stackpos]^.actransl^.transla = 2) then
              presentstate := 'N';
          end;
    'C','D' : cdtransition;
              (* transition from states 'C' and 'D' *)
    'E'     : etransition;
              (* transition from state 'E' *)
    'F'     : ftransition;
              (* transition from state 'F' *)
    'G'     : gtransition;
              (* transition from state 'G' *)
    'H'     : htransition;
              (* transition from state 'H' *)
    'I'     : itransition;
              (* transition from state 'I' *)
    'J'     : jtransition;
              (* transition from state 'J' *)
    'M'     : mtransition;
              (* transition from state 'M' *)
  end (* case *);
if eof(inp) then
  if (presentstate in ['C','D','E','G','M','K']) or
     ((presentstate = 'I') and (wlength > 4)) then
    presentstate := 'K'
  else
    presentstate := 'N';
sref^.presstate := presentstate;
end (* gettransition *);
```

```
procedure contract;
 (* Procedure for performing contractions, whenever
    they are detected. *)
 var
   position,sdepth,tot,i,j,k : integer;
   initrans,trans            : translref;
   temp                      : stackref;
 procedure clearupperstacks;
 var
   i,j : integer;
 begin (* clearupperstacks *)
 for i := stackpos+1 to stacklevel-decr do
   begin
   sdepth := stacks[i]^.depth;
   for j := 1 to sdepth do
     pop(stacks[i]);
   stacks[i] := stacks[i+decr];
   stacks[i+decr] := nil
   end;
 if (stacklevel-decr) >= (stackpos+1) then
   stacklevel := stackpos + decr;
 end (* clearupperstacks *);

 procedure gettrnsl(var i,j : integer; var trans : translref);
 (* Procedure for getting the translation from the dictionary
    and placing it on the current stacks or output array, as
    it corresponds *)
 var
   k : integer;
 begin (* gettrnsl *)
 j := 0;
 while trans <> nil do
   begin
   if stackpos = i then
     numtrans := succ(numtrans);
   if (stackpos=1) and (presentstate='K') then
     begin
     j := succ(j);
     word[wlength+j] := trans^.transla
     end
   else
     begin
     if (stackpos <> i) or (presentstate <> 'K') then
       begin
       push(stacks[i]);
       sref^.next := stacks[i]^.next;
       stacks[i]^ := sref^;
       with stacks[i]^ do
         begin
         depth := depth + 1;
         brreps := trans^.transla
         end;
       sref^ := stacks[i]^
       end
     end;
   trans := trans^.ntransla
   end (* while *);
 wlength := wlength + j;
 end (* gettrnsl *);
```

```
begin (* contract *)
clearupperstacks;
if decr = 0 then
  contatoutlevel := true;
sdepth := stacks[stackpos]^.depth;
initrans := sref^.actransl;
if decr = 0 then
  sref^.numichars := sref^.numichars + 1;
for i:= stackpos downto 1 do
  begin
  trans := initrans;
  if i < stackpos then
    sref^ := stacks[i]^;
  if decr > 0 then
    begin
    if stacks[stackpos+1] = nil then
      tot := decr
    else
      tot := decr - stacks[stackpos+1]^.depth
    end
  else
    tot := 0;
  k := 0;
  for j := 1 to sdepth do
                (* pop chars that will be contracted *)
    begin
    if (decr > 0) and (stacks[stackpos+1]^.depth <= j) then
      begin
      if k < tot then
        word[wordsize-k] := stacks[i]^.brreps;
      k := succ(k)
      end;
    pop(stacks[i])
    end;
  if stacks[i] <> nil then
                (* store present stack depth after pop *)
    sref^.depth := stacks[i]^.depth
  else
    sref^.depth := 0;
  gettrnsl(i,j,trans)
        (* get translation for current contraction *)
  end (* for *);
if presentstate = 'K' then
  begin
  if (decr > 0) and (stackpos = 1) then
    begin
    for j := 1 to tot do
      word[wlength+j] := word[wordsize-tot+j];
    wlength := wlength + tot
    end;
  for i := stackpos to stacklevel - 1 do
    begin
    stacks[i] := stacks[i+1];
    if stacks[i] <> nil then
      stacks[i]^.level := stacks[i]^.level - 1;
    stacks[i+1] := nil
    end;
```

```
  stacklevel := pred(stacklevel);
  end
else  (* presentstate = 'M' *)
  begin
  if (decr > 0) and (stackpos = 1) then
    begin
    for j := 1 to tot do
      begin
      push(stacks[stackpos]);
      sref^.next := stacks[stackpos]^.next;
      stacks[stackpos]^ := sref^;
      with stacks[stackpos]^ do
        begin
        depth := succ(depth);
        brreps := word[wordsize - tot + j]
        end;
      sref^ := stacks[stackpos]^;
      end;
    getposatnextcont;
    if presentstate = 'M' then
      begin
      sref^.acichars := sref^.acichars^.nexttransi;
      presentstate := sref^.acichars^.state
      end;
    if presentstate = 'N' then
      begin
      push(stacks[stackpos]);
      temp := stacks[stackpos]^.next;
      with stacks[stackpos]^ do
        begin
        level := temp^.level;
        posatbw := temp^.posatbw;
        depth := temp^.depth;
        acnode := temp^.acnode;
        accont := temp^.accont;
        acichars := temp^.acichars;
        numichars := temp^.numichars;
        actransi := temp^.actransi;
        actransl := temp^.actransl
        end
      end
    else
      begin
      push(stacks[stackpos]);
      sref^.next := stacks[stackpos]^.next;
      sref^.numichars := sref^.numichars + 1;
      stacks[stackpos]^ := sref^
      end;
    with stacks[stackpos]^ do
      begin
      depth := succ(depth);
      brreps := brrep;
      chs := inputch;
      if presentstate = 'N' then
        presstate := 'N';
      end
    end
  end
```

```
end (* contract *);

procedure donotcontract;
(* Procedure for performing actions associated with the
   decision of not contracting. *)
var
  position,sdepth,i,j,w1,w2 : integer;
  asg                       : boolean;
begin (* donotcontract *)
sdepth := topofstack^.depth;
for i := sdepth downto 1 do
  begin
  if stackpos = 1 then
    word[wlength + i] := topofstack^.brreps;
  pop(topofstack)
  end;
stacks[stackpos] := nil;
for i := stackpos to stacklevel - 1 do
  begin
  stacks[i] := stacks[i + 1];
  if stacks[i] <> nil then
    stacks[i]^.level := stacks[i]^.level - 1;
  stacks[i + 1] := nil
  end;
stacklevel := pred(stacklevel);
if stackpos = 1 then
  begin
  if (inputch in ['A'..'Z']) then
    wordcomplete := false
  else
    wordcomplete := true;
  if stacks[1] <> nil then
    begin
    if contatoutlevel then
      wlength := wlength + sdepth - stacks[1]^.depth
    else
      wlength := wlength + sdepth - stacks[1]^.depth + 1
    end
  else
    wlength := wlength + sdepth;
  if blankgen then
    begin
    wlength := succ(wlength);
    if (decr = 0) or (wlength = 2) then
      word[wlength] := 0
    else
      begin
      for i := wlength - 1 downto nletters + 1 do
        word[i+1] := word[i];
      word[nletters+1] := 0
      end
    end
  end;
end (* donotcontract *);

procedure postpone;
(* Procedure for performing the actions related to the
   postponement of the decision to contract or not *)
```

```
begin (* postpone *)
if inputch <> ' ' then
  begin
  sref^.numichars := sref^.numichars + 1;
  if not contatoutlevel then
          (* no previous contraction at an outer level *)
    begin
    if sref^.acichars <> nil then
      sref^.acichars := sref^.acichars^.nexttransi;
    sref^.depth := sref^.depth + 1;
    push(topofstack);
    topofstack^ := sref^
    end
  else
    begin
    sref^.next := topofstack^.next;
    topofstack^ := sref^;
    with topofstack^ do
      if acichars <> nil then
        acichars := acichars^.nexttransi;
    end;
  stacks[stackpos] := topofstack
  end
end (* postpone *);


  begin (* translate *)
  decr := 0;
  blankgen := false;
  possiblecont := true;
  topofstack := stacks[stackpos];
  brrepsinordbchset := topofstack^.brreps in ordbchset;
  if topofstack^.actransi = nil then
    begin
    if not brrepsinordbchset then
      begin
      if not (inputch in delims) and
         not (inputch in numbers) then
        getposatnextcont
      else
        begin
        presentstate := 'N';
        possiblecont := false
        end
      end
    else
      presentstate := ' '
    end
  else
    presentstate := topofstack^.actransi^.state;
  if (topofstack^.depth = 1) and brrepsinordbchset then
    begin
    x := getkey(topofstack,inputch);
    if inputch in ['A'..'Z'] then
      possiblecont := checkdigram(topofstack,inputch)
    else
      possiblecont := false;
```

```
  if topofstack^.posatbw and (inputch in ['A'..'Z']) then
    begin
    exceptions := excepdigram[ord(topofstack^.chs)
                              - ord('A') + 1, ord(inputch)
                              - ord('A') + 1];
    xx := x
    end;
  if possiblecont then
    begin
    with sref^ do
      begin
      level := stackpos;
      chs := inputch;
      brreps := brrep;
      posatbw := topofstack^.posatbw;
      depth := 1;
      presstate := presentstate;
      acnode := locatekey(x,root);
      accont := acnode^.contra;
      acichars := accont^.ichars^.nexttransi;
      numichars := 1;
      if posatbw then
        begin
        if accont^.transibw <> nil then
          actransi := accont^.transibw
        else
          actransi := nil
        end
      else
        begin
        if accont^.transinbw <> nil then
          actransi := accont^.transinbw
        else
          actransi := nil
        end;
      actransl := accont^.transl;
      next := topofstack
      end
    end
  end
else
  begin
  sref^ := topofstack^;
  sref^.chs := inputch;
  if not contatoutlevel then
    sref^.brreps := brrep;
  sref^.next := topofstack
  end;
if possiblecont then
        (* there is possibility of contracting *)
  begin
  gettransition;
  if presentstate in ['M','K'] then
    contract
  else
    if presentstate ='N' then
      donotcontract
    else
```

```
      postpone
  end
else
  donotcontract
end (* translate *);

begin (* formbraword *)
if gradeii and not bypasscontraction then
  begin
  numtrans := 0;
  contatoutlevel := false;
  for stackpos := stacklevel + 1 downto 1 do
    begin
    if stackpos <> 1 then
      positionatbw :=
          stacks[stackpos-1]^.presstate in ['I','J'];
    if stacks[stackpos] = nil then
                  (* there is nothing in stack *)
      begin
      checkfornums;
      if (inputch in ['A'..'Z']) and not letterflag then
        begin
        stacklevel := succ(stacklevel);
        wordcomplete := false;
        with sref^ do
          begin
          level := stackpos;
          chs := inputch;
          brreps := brrep;
          posatbw := positionatbw;
          depth := 1;
          presstate := ' ';
          acnode := nil;
          accont := nil;
          acichars := nil;
          numichars := 1;
          actransi := nil;
          actransl := nil;
          next := nil
          end;
        push(topofstack);
        topofstack^ := sref^;
        stacks[stackpos] := topofstack
        end
      else
        begin
        if stackpos = 1 then
          begin
          if inputch in numbers then
            begin
            wlength := succ(wlength);
            word[wlength] := brrep;
            wordcomplete := false
            end
          else
            if not posblankcont then
              wordcomplete := true
          end
```

```
        end
      end
    else  (* there is something in stack *)
      begin
      sref^ := stacks[stackpos]^;
      translate
      end;
    if (stackpos = 1) and (inputch in delims) and
     (not posblankcont) then
      if (inputch in [''',',"',',#',',(',',[',',&']) or
        (brrep = 4) then
        begin
        wordcomplete := false;
        wlength := succ(wlength);
        word[wlength] := brrep
        end
      else
        if presentstate = 'J' then
          wordcomplete := (wlength > 0)
        else
          wordcomplete := true;
    if eof(inp) and (stackpos = 1) then
      wordcomplete := true;
    if wordcomplete then
      begin
      positionatbw := true;
      if inputch in numbers then
        checkfornums;
      wlength := succ(wlength);
      word[wlength] := brrep;
      if (inputch in ['[',']','%']) or
         ((inputch = '*') and (brrep = 20)) then
        begin
        wlength := succ(wlength);
        word[wlength] := auxbrrep
        end
      end
    end
  end
else
  begin
  if stacks[1] <> nil then
    begin
    stackpos := 1;
    sref^ := stacks[1]^;
    translate
    end;
  checkfornums;
  wlength := succ(wlength);
  word[wlength] := brrep;
  wordcomplete := (inputch in delims) and not
                  (brrep = 4) and not
                  (inputch in [''',',"',',#',',&',',(',',['])
  end
end (* formbraword *);
```

```
procedure formbraline;
 (* procedure for forming the output Braille line *)

 var
   i,j : integer;
 procedure lookforexcep;
 (* procedure for searching for exceptions *)
 var
   change,getout,chlast : boolean;
   i,j,k                : integer;
 begin
 change := false;
 getout := false;
 chlast := false;
 with sref^ do
   begin
   acnode := locatekey(xx,exceproot);
   accont := acnode^.contra;
   acbraillerep := accont^.braillerep;
   actransl := accont^.transl;
   i := 0;
   k := 0;
   while (not change) and (not getout) do
     begin
     i := succ(i);
     if acbraillerep^.transla = word[i] then
       acbraillerep := acbraillerep^.ntransla
     else
       begin
       if (((word[i] = 43) or (word[i] = 59)) and
                                          (* ED & ER *)
          (acbraillerep^.transla = 17)) then
         begin
         change := true;
         chlast := true;
         for j := i to wlength do
           begin
           word[wordsize-j+i] := word[j];
           k := succ(k);
           end;
         if (xx = 447) and (word[2] = 34) then
                        (* exception for 'renamed' *)
           begin
           word[wordsize] := 25; (* Braille char for 'D' *)
           word[wordsize - 1] := word[wlength];
           k := 2;
           chlast := false
           end
         end
       else
         if acbraillerep^.transla = 0 then
           begin
           if (word[i] in ordbchset) or
              (word[i] in [2,34,43,59]) then
             getout := true;
           end
         else
           getout := true;
       end;
```

```
    if (acbraillerep = nil) and not getout then
      begin
        change := true;
      if i < wlength then
        begin
        for j := i + 1 to wlength do
          begin
          word[wordsize-j+i+1] := word[j];
          k := succ(k)
          end
        end
      end
    else
      if (acbraillerep^.transla = 0) and not getout then
        if word[i+1] in ordbchset then
          getout := true
        else
          begin
          change := true;
          for j := i + 1 to wlength do
            begin
            word[wordsize-j+i+1] := word[j];
            k := succ(k)
            end;
      if getout then
        if accont^.nextcontraction <> nil then
          begin
          getout := false;
          i := 0;
          accont := accont^.nextcontraction;
          acbraillerep := accont^.braillerep;
          actransl := accont^.transl;
          end
      end (* while *);
  if change then
    begin
    i := 0;
    while actransl <> nil do
      begin
      i := succ(i);
      word[i] := actransl^.transla;
      actransl := actransl^.ntransla
      end;
    if chlast then
      i := pred(i);          (* last char is eliminated *)
    for j := i+1 to i+k do
      word[j] := word[wordsize-j+i+1];
    wlength := i + k;
    end (* if *)
  end (* with *)
end (* lookforexcep *);

procedure generalexcep;
(* Procedure for treating general exceptions
   of letter combinations *)
begin (* generalexcep *)
if (word[1] = 20) and (inputch in delims) and
```

```
    (inputch <> ' ') and (wlength = 2) then
  begin  (* 'IN' standing alone, followed by
            punctuation marks *)
    word[3] := word[2];
    word[1] := 10;
    word[2] := 29;
    wlength := succ(wlength)
    end
else
    begin
    if inputch in delims then
      i := 1
    else
      i := 0;
    if wlength > 4 then
      begin   (* 'ENESS', 'INESS' 'STION' 'STHE' *)
      if (word[wlength-i] = 14) and
       (word[wlength-i-1] = 14) and
        (word[wlength-i-2] = 17) then
        begin
        if word[wlength-i-3] in [20,34] then
          begin
          word[wlength-i-2] := 48;
          word[wlength-i-1] := 14;
          if word[wlength-i-3] = 20 then
            word[wlength-i-3] := 10
          else
            word[wlength-i-3] := 17;
          word[wlength-i] := word[wlength];
          wlength := wlength - i
          end
        end
      else
        if (word[wlength-i] = 29) and
         (word[wlength-i-1] = 21) and
          (word[wlength-i-2] = 10) and
          (word[wlength-i-3] = 12) then
          begin
          word[wlength-i-3] := 14;
          word[wlength-i-2] := 48;
          word[wlength-i-1] := 29;
          word[wlength-i] := word[wlength];
          wlength := wlength - i
          end
      end;
    if (inputch in delims) and (wlength > 1) then
      if word[wlength-1] = 2 then
        begin   (* 'EA' at the end of the word *)
        word[wlength + 1] := word[wlength];
        word[wlength] := 1;
        word[wlength - 1] := 17;
        wlength := succ(wlength)
        end;
    end;
end (* generalexcep *);

begin (* formbraline *)
posblankcont := false;
```

```
nletters := 0;
if word[1] = 0 then
  begin
  while (word[1] = 0) and (wlength > 1) do
    begin
    for i := 1 to wlength - 1 do
      word[i] := word[i + 1];
    wlength := pred(wlength);
    end
  end;
generalexcep;
if wlength > 1 then
  while (word[wlength] = 0) and (word[wlength - 1] = 0) do
    wlength := pred(wlength);
if exceptions then
  begin
  lookforexcep;
  exceptions := false;
  end;
if linesize >= (linecount + wlength) then
  begin
  for i := linecount + 1 to linecount + wlength do
    line[i] := word[i - linecount];
  linecount := linecount + wlength;
  wlength := 0;
  wordfitinline := true;
  linecomplete := (linesize = linecount)
  end
else
  begin
  if wlength > linesize then
    begin
    writeln('The word length is greater than
             the line size defined.');
    wlength := linesize
    end
  else
    begin
    j := wlength;
    i := 1;
    while (i <> j+1) and (i < wlength) do
      begin
      if word[i] = 0 then
        j := i;
      i := succ(i)
      end;
    if linesize >= linecount+j then
      begin
      for i := 1 to j-1 do
        line[linecount+i] := word[i];
      linecount := linecount + j - 1;
      for i := j + 1 to wlength do
        word[i-j] := word[i];
      wlength := wlength - j
      end;
    linecomplete := true;
    wordfitinline := false
    end
```

```
    end
end (* formbraline *);

procedure outputbrailleline;
(* Physical output of a Braille line *)
const
  horbrachlen = 3;
  verbrachlen = 2;
  blank = ' ';
  dot = '.';
var
  brailledots : array[1..horbrachlen,
                      1..verbrachlen,1..linesize]
                of boolean;
  horpos      : 1..horbrachlen;
  verpos      : 1..verbrachlen;
  chpos       : integer;
procedure selectdots (brrep : integer);
  (* formation of Braille characters, according
     to mapping defined *)
  var
    powerof2 : integer;
    temp     : boolean;
  begin
  powerof2 := 32;          (* Greatest power of 2 < 63 *)
  for verpos := verbrachlen downto verbrachlen - 1 do
    for horpos := horbrachlen downto 1 do
      begin
      if (brrep - powerof2) >= 0 then
        begin
        brailledots[horpos,verpos,chpos] := true;
        brrep := brrep - powerof2
        end
      else
        brailledots[horpos,verpos,chpos] := false;
      powerof2 := powerof2 div 2
      end (* for *)
  end (* selectdots *);

begin (* outputbrailleline *)
linenumber := succ(linenumber);
for chpos := 1 to linecount do
  selectdots(line[chpos]);
for horpos := 1 to horbrachlen do
  begin
  for chpos := 1 to linecount do
    begin
    for verpos := 1 to verbrachlen do
      if brailledots[horpos,verpos,chpos] then
        write(out,dot)
      else
        write(out,blank);
    write(out,blank)
    end;
  writeln(out,blank)
  end;
if linenumber < linesperpage then
  writeln(out,blank)
```

```
else
  begin
  linenumber := 0;
  writeln(out)    (* page(out) *)
  end
end (* outputbrailleline *);

procedure preparenextline;
begin (* preparenextline *)
linecount := 0;
if wlength = 0 then
  begin
  wordcomplete := false;
  linecomplete := false
  end
end (* preparenextline *);
```

```
begin  (* program ebt *)
 writeln;
 writeln;
 writeln('        English into Braille Translation Program');
 writeln;
 openfiles;
 initialise;
 mapbrailletoascii;
 if gradeii then
   begin
   writeln;
   writeln('loading dictionary . . .');
   buildabconttransnetw;
   end;
 mark(heap);
 writeln('translating . . .');

 while not eof(inp) do
   begin
   getbraillerep(brrep,inputch);
   if not newparagraph and not bypassgradeii then
     formbraword;
   repeat
     if wordcomplete then
       formbraline;
     if linecomplete or eof(inp) or newparagraph then
       begin
       release(heap);
       outputbrailleline;
       preparenextline;
       mark(heap);
       end;
   until wordfitinline
   end;
 writeln;
 writeln('End of translation.');
 close (out,lock)
 end  (* program ebt *).
```

Appendix 7

SET OF TESTS USED FOR THE VALIDATION OF THE EBT PROGRAM

The establishment of a complete set of tests that should be included to verify the correctness of any program is a difficult task, and certainly individual to each program, depending on its purpose.

In the case of the translator being considered, it is of utmost importance to check for two things:

- that the input text is adequately translated when it ought to be.

- and that it is not translated when it ought not to be.

This is not even easy to define. However, a testing strategy has been defined and implemented, to cover the two aspects mentioned.

The test data includes not only the words that should be translated, but also the ones that should not be translated. This provides a complete view of the performance of the translation program. The tests defined are the following:

1. Test for abbreviations and contractions standing alone.

2. Tests for abbreviations and contractions not at the beginning of the word.

3. Tests for abbreviations and contractions in the middle of the word.

4. Tests for showing the choice of contractions by the translator.

5. Tests for exceptions to the general rules.

6. Tests for space contractions.

7. Tests for translation of punctuation marks.

8. Tests for translation of numbers.

With the first 5 tests in mind, a table was designed to represent in the clearest way possible, the behaviour of English Braille Grade II contractions and abbreviations (Table A.7.1).

The first column of this table contains a complete alphabetically ordered list, of all of the English Grade II Braille contractions and abbreviations (BCAs). This column serves for testing all the BCAs that stand alone; the ones

that cannot exist standing alone are marked with an asterisk. The second column lists the words that contain BCAs at the beginning of the word. The third column contains BCAs occurring not at the beginning of the word, and the fourth, BCAs occurring in the middle of the word. The fifth column contains examples related to choice of contractions between conflicting contractions and/or exceptions for each case in which these exist. The empty cells in the table indicate that no words were found in English that contained exactly the particular contraction or abbreviation, while satisfying the position restrictions imposed for each column. This table is presented next.

WORDS STANDING ALONE	TEST AT THE BEGINNING OF THE WORD	TEST NOT AT THE BEG. OF THE WORD	TEST IN THE MIDDLE OF THE WORD	CHOICE OF CONTRAC. EXCEPTIONS
ABOUT			WHEREABOUTS	
ABOVE				
ACCORDING	ACCORDINGLY			
ACROSS				
AFTER	AFTERS			
AFTERNOON				
AFTERWARD				
AFTERWARDS				
AGAIN				
AGAINST				
ALLY		NATURALLY		
ALMOST				
ALREADY				
ALSO				
ALTHOUGH				
ALTOGETHER				
ALWAYS				
ANCE *	ANCESTOR	ENDURANCE	INSTANCES	ENHANCED
				DANCER
AND	ANDY	HAND	BANDIT	
AR *	ARM	BAR	WARM	EAR
				BEARD
AS	ASK	HAS	BASS	
ATION *		INFORMATION	INTERNATIONAL	

Set of Tests for Braille Translator

Table A.7.1

WORDS STANDING ALONE	TEST AT THE BEGINNING OF THE WORD	TEST NOT AT THE BEG. OF THE WORD	TEST IN THE MIDDLE OF THE WORD	CHOICE OF CONTRAC. EXCEPTIONS
BB *			RABBIT	ABBEY
				COBBLER
BE	BELONG	BRIBE	ABERRATION	BEEN BELL
				BETTER BED
				BEAVER
BECAUSE				
BEFORE	BEFOREHAND			
BEHIND				
BELOW				
BENEATH				
BESIDE				
BESIDES				
BETWEEN				
BEYOND				
BLE *	BLESS	ABLE	ENABLES	DISABLED
				DOUBLED
				GAMBLER
BLIND	BLINDING			BLINDNESS
BRAILLE				
BUT	BUTTER	DEBUT	ABUTMENT	
BY	BYLAW	LULLABY		
CAN	CANDY	PECAN	DECANT	
CANNOT				
CC *			ACCEPT	ACCOMPLISH
CH *	CHOIR	BEACH	ECHO	

Table A.7.1 (continued)

WORDS STANDING ALONE	TEST AT THE BEGINNING OF THE WORD	TEST NOT AT THE BEG. OF THE WORD	TEST IN THE MIDDLE OF THE WORD	CHOICE OF CONTRAC. EXCEPTIONS
CHARACTER	CHARACTERISTIC			
CHILD	CHILDHOOD	ROTHSCHILD		
CHILDREN				
COM *	COMPUTE		UNCOMFORTABLE	COMB
CON *	CONCENTRATE	SILICON	ECONOMY	CONK
				CONES
CONCEIVE	CONCEIVED			CONCEIVABLE
CONCEIVING	CONCEIVINGLY			
COULD	COULDN'T			
DAY	DAYS	MONDAY		
DD *		ADD	ADDS	BEDDING
DECEIVE	DECEIVED			
DECEIVING	DECEIVINGLY			
DECLARE	DECLARED			
DECLARING				
DIS *	DISPLAY		UNDISMAYED	DISC
				DISTURB
				DISHES
DO	DONE	IDOLATRY		
EA *	EAGLE	TEA	LEAD	LEARN
				FEAR
ED *	EDITOR	TRUSTED	EMBEDDED	PREDETERMINE
				LOATHED
EITHER				
EN *	ENTRANCE	HEN	PENNY	THEN

Table A.7.1 (continued)

WORDS STANDING ALONE	TEST AT THE BEGINNING OF THE WORD	TEST NOT AT THE BEG. OF THE WORD	TEST IN THE MIDDLE OF THE WORD	CHOICE OF CONTRAC. EXCEPTIONS
ENCE *		PESTILENCE	EXPERIENCES	EXPERIENCED
ENOUGH				
ER *	ERRADICATE	LARGER	PERFORM	OTHER
EVER	EVERYBODY	FEVER		SEVERE
				PERSEVERE
				REVERSE
EVERY	EVERYTHING			
FATHER	FATHERLY			
FF *		CLIFF	CLIFFS	EFFORT
				OFFER
FIRST	FIRSTLY			
FOR	FOREST		INFORM	
FRIEND	FRIENDLY	BEFRIEND	UNFRIENDLY	
FROM	FROMAGE			
FUL *	FULLY	RESPECTFUL	FRIGHTFULNESS	
GG *			RAGGED	
GH *	GHOST	SIGH	EIGHT	
GO	GONE	EGO	EGOISM	
GOOD	GOODNESS			
GREAT	GREATER			
HAD	HADDOCK			
HAVE	HAVEN	BEHAVE	BEHAVED	
HERE	HEREWITH	ADHERE	ADHERED	HERETIC
				SPHERE
HERSELF				

Table A.7.1 (continued)

WORDS STANDING ALONE	TEST AT THE BEGINNING OF THE WORD	TEST NOT AT THE BEG. OF THE WORD	TEST IN THE MIDDLE OF THE WORD	CHOICE OF CONTRAC. EXCEPTIONS
HIM				
HIMSELF				
HIS	HISS	THIS		
IMMEDIATE	IMMEDIATELY			
IN	INSIGHT	PIN	DINNER	
ING *	INGRATE	SINGING	NIGHTINGALE	
INTO	INTOLERANT	PINTO		
IT	ITERATE	BANDIT	SPITTED	
ITS		BANDITS		
ITSELF				
ITY *		ENTITY		FRUITY
JUST	JUSTIFY	ADJUST	ADJUSTMENT	
KNOW	KNOWN			
KNOWLEDGE	KNOWLEDGEABLE	ACKNOWLEDGE	ACKNOWLEDGEMENT	
LESS *	LESSON	TOPLESS	BLESSING	
LETTER	LETTERS			
LIKE	LIKELY	DISLIKE	DISLIKED	
LITTLE				
LORD	LORDLY			
MANY		GERMANY		
MENT *	MENTAL	INCREMENT	ELEMENTS	
MORE	MOREOVER	SOPHOMORE	SOPHOMORES	
MOTHER	MOTHERLY	SMOTHER		
MUCH				
MUST	MUSTANG			

Table A.7.1 (continued)

WORDS STANDING ALONE	TEST AT THE BEGINNING OF THE WORD	TEST NOT AT THE BEG. OF THE WORD	TEST IN THE MIDDLE OF THE WORD	CHOICE OF CONTRAC. EXCEPTIONS
MYSELF				
NAME	NAMELY	SURNAME		RENAME
NECESSARY		UNNECESSARY		
NEITHER				
NESS		HELPLESSNESS		BARONESS
				LIONESS
NOT	NOTHING	DENOTE		
OF	OFFICE		COFIN	
ONE	ONEROUS	PRONE	BONES	COLONEL
				TONED
				ANEMONE
				PIONEER
ONESELF				
ONG *		WRONG	LONGER	CONGRESS
OU *	OURS		COUGH	
OUGHT		BOUGHT	THOUGHTFUL	
OUND *		COMPOUND	FOUNDER	
OUNT *		COUNT	MOUNTAIN	
OURSELVES				
OUT	OUTSIDE	SPROUT	ROUTE	
OW *	OWNER	NOW	FLOWER	
PAID		UNPAID		
PART	PARTLY	COUNTERPART	DEPARTED	
PEOPLE				
PERCEIVE	PERCEIVES			PERCEIVABLE

Table A.7.1 (continued)

WORDS STANDING ALONE	TEST AT THE BEGINNING OF THE WORD	TEST NOT AT THE BEG. OF THE WORD	TEST IN THE MIDDLE OF THE WORD	CHOICE OF CONTRAC. EXCEPTIONS
PERCEIVING				
PERHAPS				
QUESTION	QUESTIONABLE			
QUICK	QUICKLY			
QUITE				
RATHER				
RECEIVE	RECEIVED			RECEIVABLE
RECEIVING				
REJOICE	REJOICED			
REJOICING				
RIGHT	RIGHTHANDED	BRIGHT	FRIGHTFUL	
SAID				
SH *	SHORE	MESH	RUSHED	
SHALL	SHALLOW	MARSHAL		
SHOULD	SHOULDN'T			SHOULDER
SION *	SIONISM	EXTENSION	PASSIONAL	
SO	SOLE		ASSOCIATE	
SOME	SOMEBODY	HANDSOME		BLOSSOMED
SPIRIT	SPIRITUOUS			
ST *	STAND	LOST	RASTER	
STILL	STILLNESS			
SUCH				
TH *	THIN	OATH	ETHICS	
THAT	THATCHER			
THE	THEME	LOATHE	ATHEIST	THERMAL

Table A.7.1 (continued)

WORDS STANDING ALONE	TEST AT THE BEGINNING OF THE WORD	TEST NOT AT THE BEG. OF THE WORD	TEST IN THE MIDDLE OF THE WORD	CHOICE OF CONTRAC. EXCEPTIONS
				CATHEDRAL
				THENCE
THEIR	THEIRS			
THEMSELVES				
THERE	THEREFORE			
THESE	THESES			
THIS				
THOSE				
THROUGH	THROUGHOUT			
THYSELF				
TIME	TIMELY	OVERTIME		CENTIMETRE
TION *		MOTION	EMOTIONAL	
TO	TOAST		ATOM	
TODAY				
TOGETHER		ALTOGETHER		
TOMORROW				
TONIGHT				
UNDER	UNDERNEATH	BLUNDER	THUNDERS	UNDERIVED
UPON		COUPON		
US	USE	BUS	FUSE	
VERY				
WAS	WASP		SWASH	
WERE	WEREN'T			
WH *	WHISTLE			
WHERE	WHEREABOUTS	EVERYWHERE		WHEREVER

Table A.7.1 (continued)

WORDS STANDING ALONE	TEST AT THE BEGINNING OF THE WORD	TEST NOT AT THE BEG. OF THE WORD	TEST IN THE MIDDLE OF THE WORD	CHOICE OF CONTRAC. EXCEPTIONS
WHICH	WHICHEVER			
WHOSE				
WILL	WILLING		UNWILLING	
WITH	WITHDRAW	THEREWITH		
WORD	WORDY	FOREWORD		SWORD
WORK	WORKPEOPLE	NETWORK	NETWORKS	
WORLD	WORLDLY		UNWORLDLY	
WOULD	WOULDN'T			
YOU	YOUTH	BAYOU		
YOUNG	YOUNGSTER			
YOUR	YOURS			
YOURSELF				
YOURSELVES				

Table A.7.1 (continued)

Tests for Space Contraction

The tests for space contraction are also designed to check for cases where it should and should not be applied, according to rules 21 and 23 of the Standard English Braille (*). This set of tests is presented separately because complete sentences or phrases are needed to check for space contraction. The sentences selected are:

1. And for the withdrawal of the troops.
2. For the formation of a workplan.
3. And with a great offer.
4. For her, for the girl and for the rest of the people.
5. It was given to him by the beach.
6. He went into the cave with a knife.
7. They passed by.
8. He was referred to, when they were speaking.

In cases 1 to 4, space contraction between the words 'A', 'AND', 'FOR', 'OF', 'THE', 'WITH' is shown, including cases where the space contraction could be mistakenly performed. Cases 5 and 6 show situations where the space after 'TO',

---

(*) Rule 21.- The word signs "a", "and", "for", "of", "the", "with" may follow one another without the space between them, when the sense permits. The word signs should be used as part of words wherever possible in preference to any other contraction, unless their use entails waste of space.

Rule 23.- The contractions for "to", "into" and "by" are always to be written close up to the word which follows. They may never be joined to other words by the hyphen to form compound words. They may be contracted before the numeral, capital, italic, letter and accent signs, but not before any other Braille composition or punctuation sign.

'BY', 'INTO' should be contracted; cases 7 and 8 show when the space should not be contracted.

Tests for Translation of Punctuation Marks

Most of the punctuation marks are translated in a straightforward way; however, there are some exceptions in the cases of adjacency with lower signs (signs without dots 1 or 4 in the Braille cell). The tests performed were on words followed by a space, and then the same words followed by different punctuation marks. Both, straightforward cases and exceptions are included in the following list.

1. About about; above above, according according.
2. His his; his? his-
3. Was was. was? was-
4. Be be, be? be.
5. Were were? were. were-
6. In in, in.
7. Enough enough, enough!
8. Childlike, child-like.
9. Can't, don't, couldn't, shouldn't, child's

In case 1, there is no difference in translation with an adjacent punctuation mark. However, cases 2 to 5 show words which are not allowed adjacent puntuaction marks. Cases 6 and 7 show the exceptions in the translation of some lower wordsigns. In case 8, 'childlike' is completely spelled

out, whereas in 'child-like', both contractions for 'child' and 'like' may be used. Case 9 shows the use of apostrophe with some words.

Tests for Translation of Numbers

The tests for the translation of numbers involves numbers by themselves (1), numbers in relation to letters and operation signs as is in equations (2 and 3), and numbers in relation to text (4 and 5). The list of tests performed is given next.

1. 23 35 48 59 1123
2. 23a+4b=43;
3. 45z+11y-35x*23w=15
4. 100 by 22. 2 to 3.
5. Year of 1983.

The set of tests presented is considered to be complete and representative for the analysis of the performance of the translation program, since most of the rules given in the Standard English Braille are included, with the exceptions noted in chapter 6.

Braille Listings of the tests performed

These listings are included in figures A.7.1 to A.7.4 for completeness, and to provide the possibility of validating the program output.

The input to the program was in the following order:

1. The words appearing in Table A.7.1 (row by row).
2. The sentences for testing space contraction.
3. The tests for the translation of punctuation marks.
4. The tests for the translation of numbers.

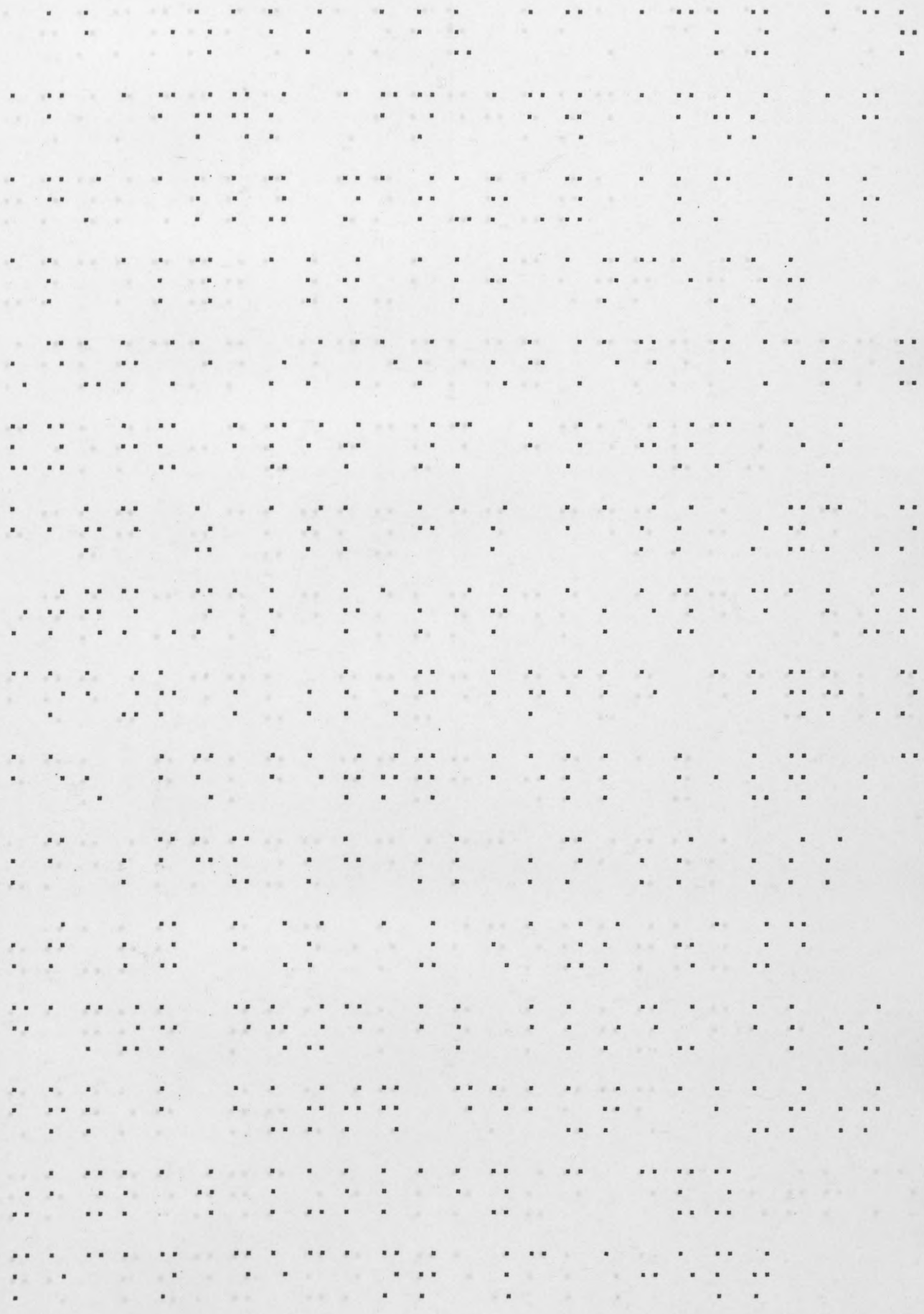


Figure A.7.1
Words appearing in table A.7.1 (row by row)

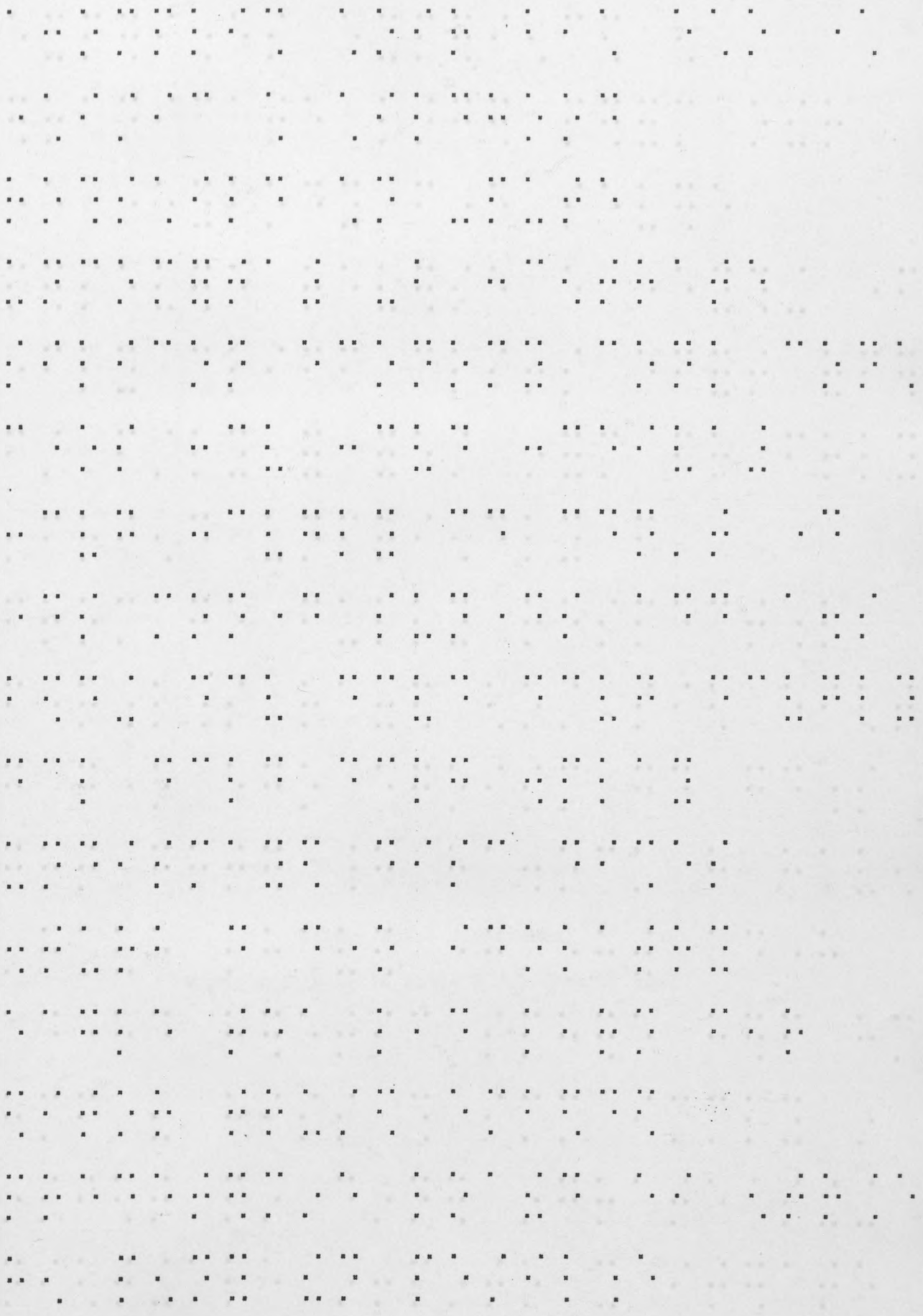


Figure A.7.1 (continued)

Words appearing in table A.7.1 (row by row)

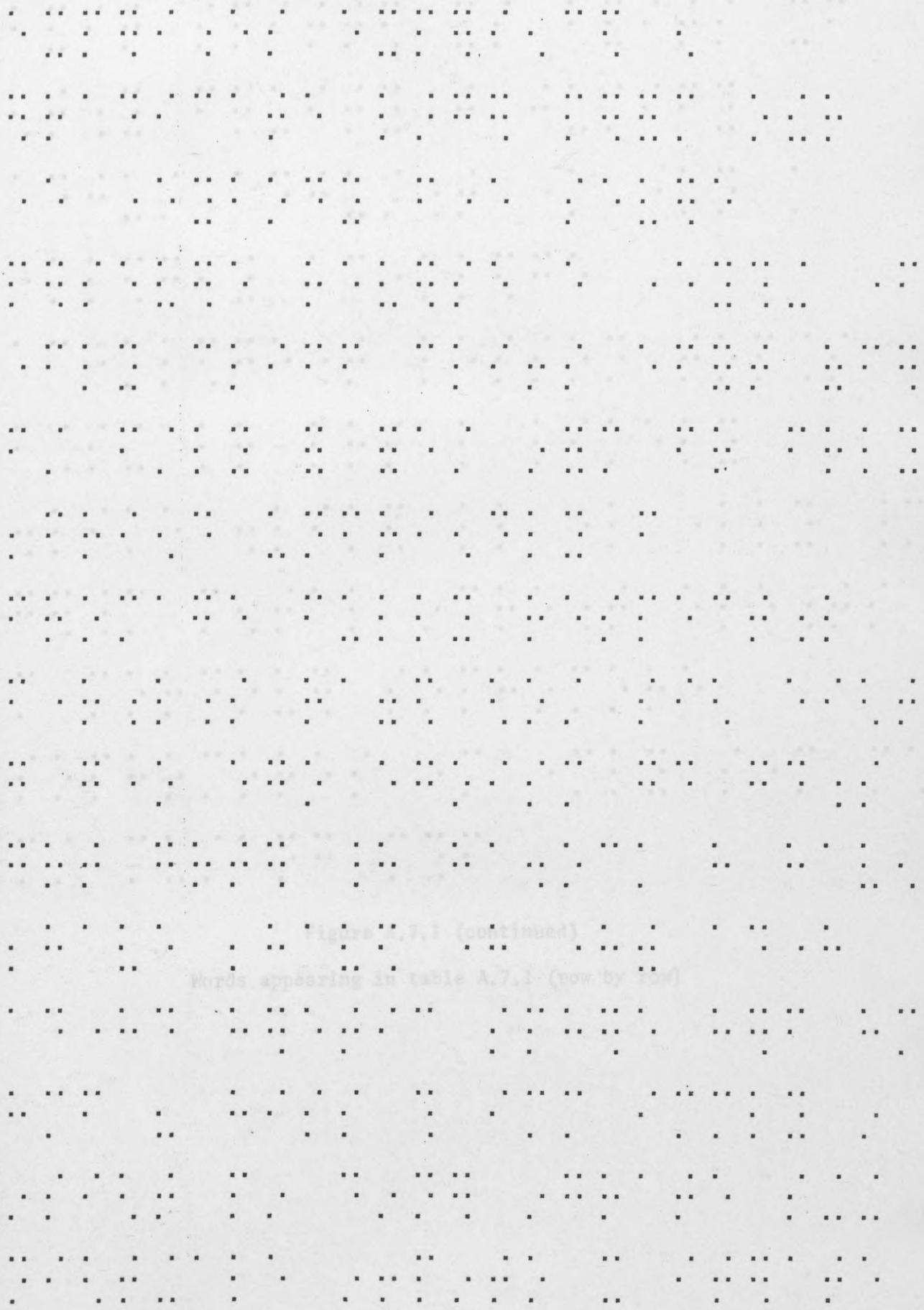


Figure A.7.1 (continued)

Words appearing in table A.7.1 (row by row)

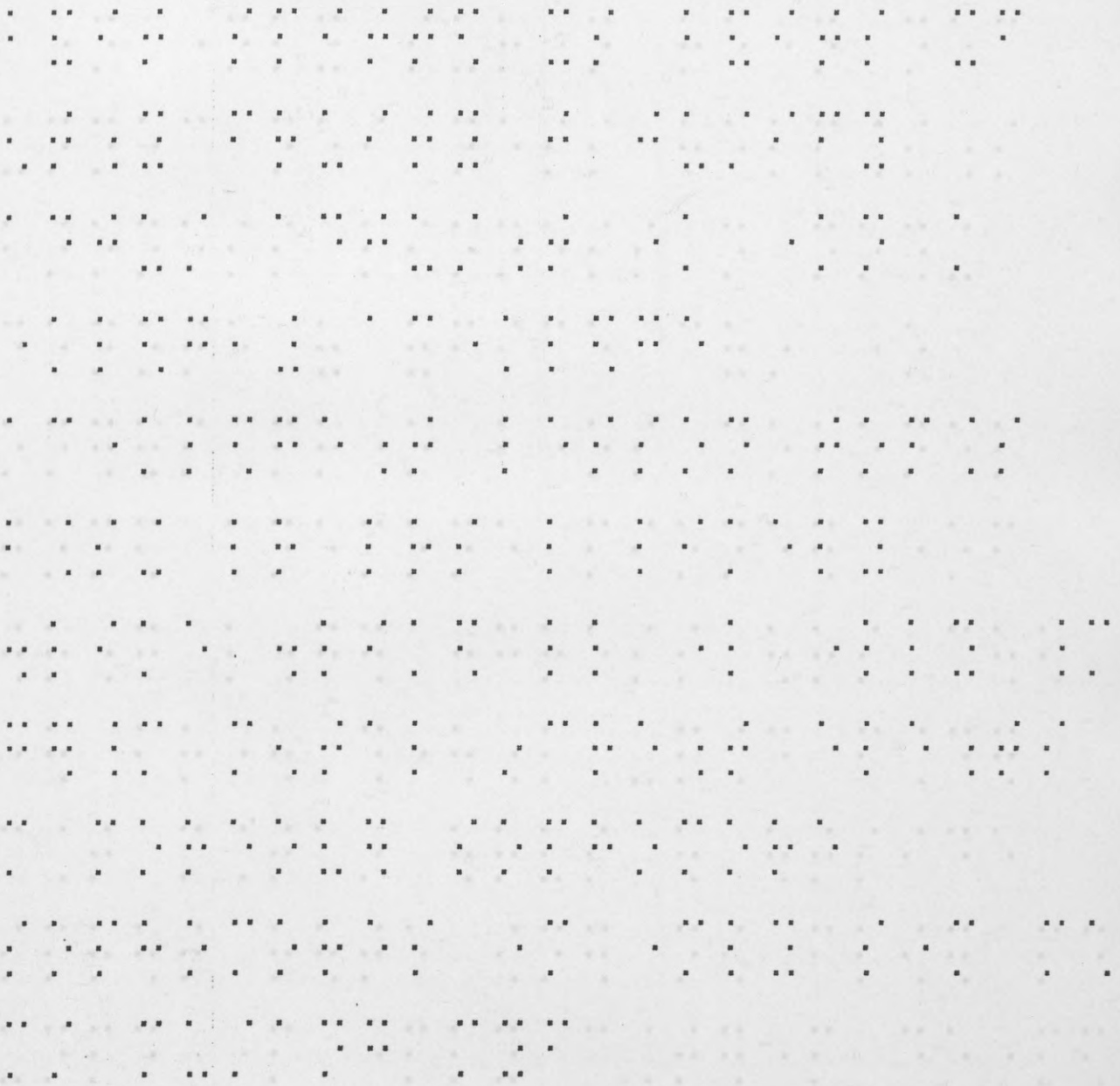


Figure A.7.1 (continued)

Words appearing in table A.7.1 (row by row)

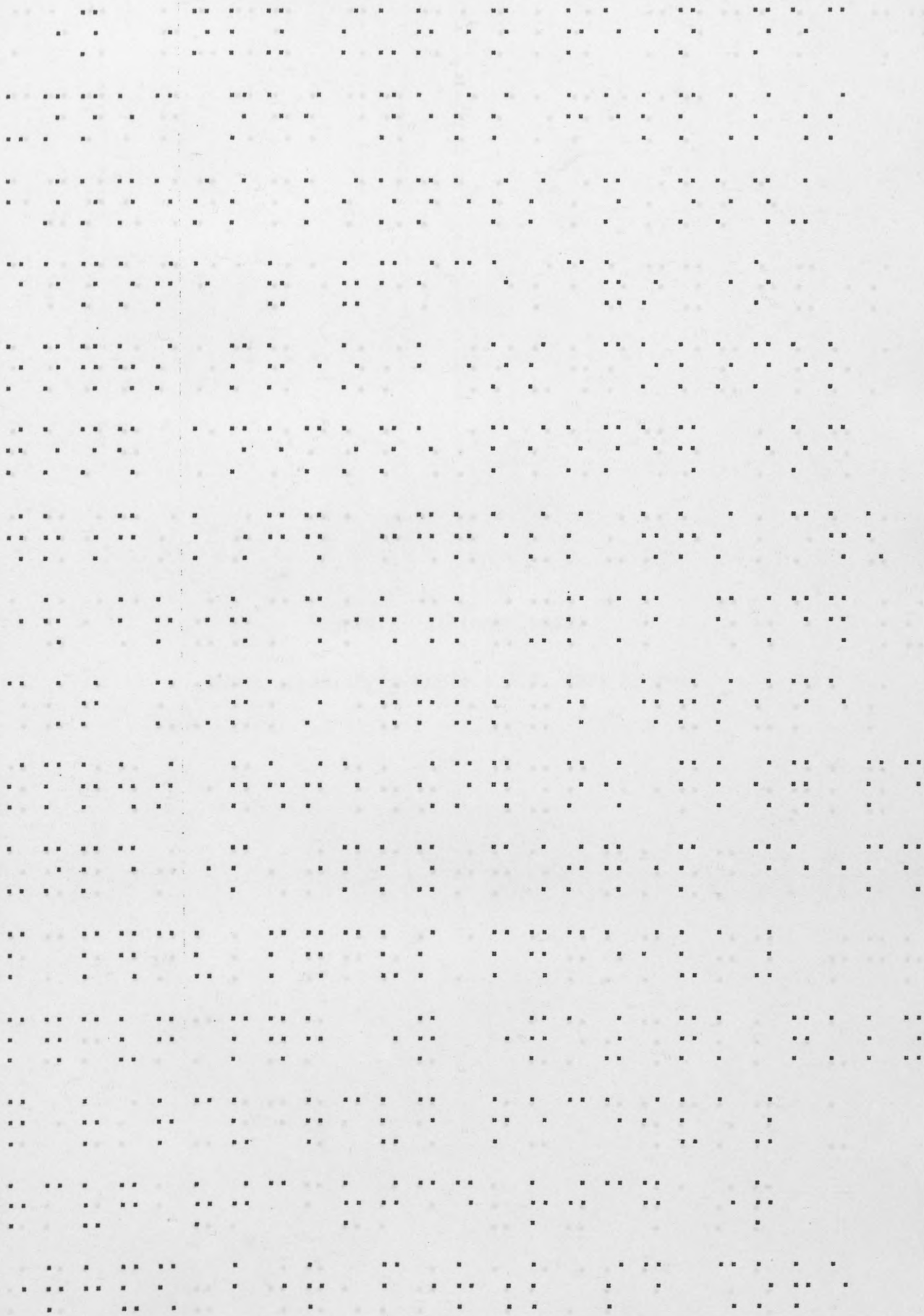


Figure A.7.1 (continued)

Words appearing in table A.7.1 (row by row)

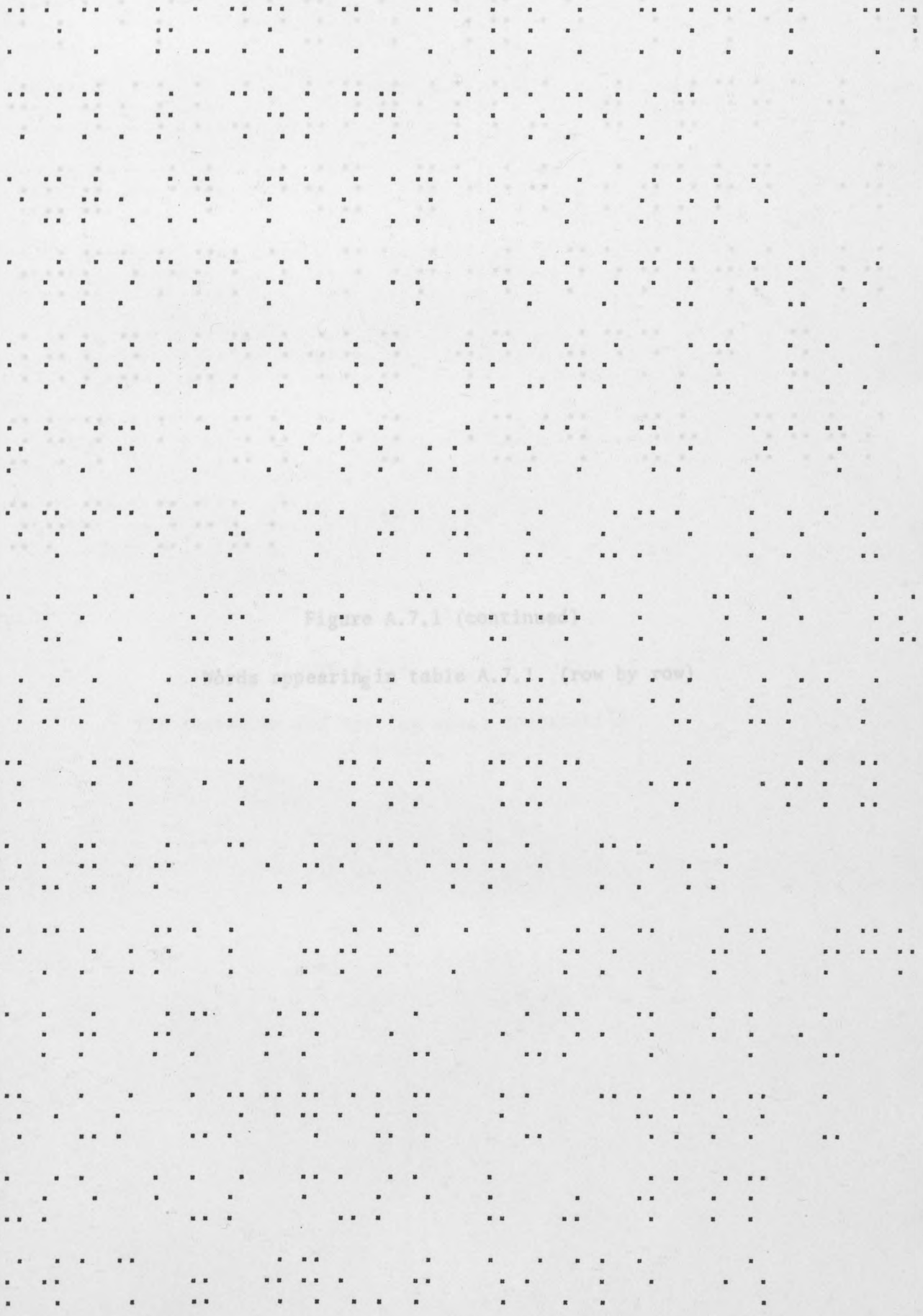


Figure A.7.1 (continued)

Words appearing in table A.7.1 (row by row)

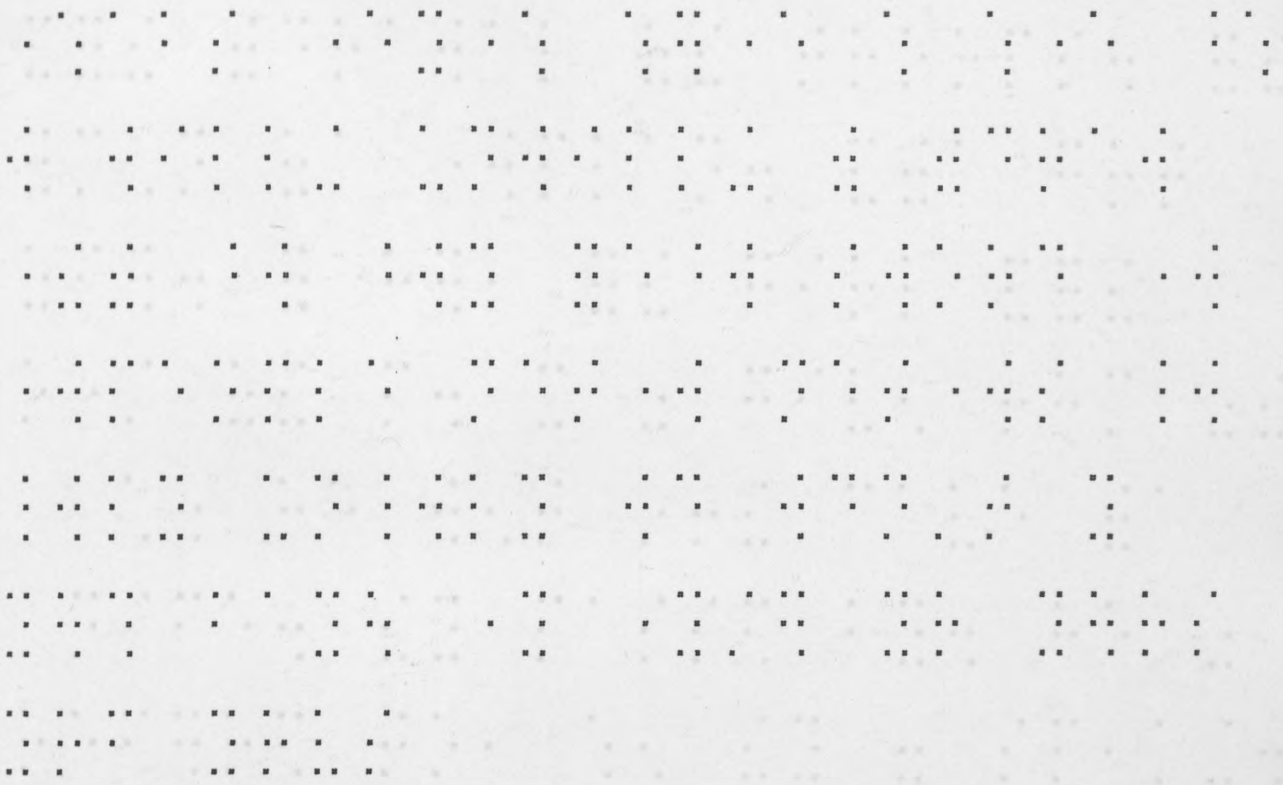


Figure A.7.1 (continued)

Words appearing in table A.7.1 (row by row)

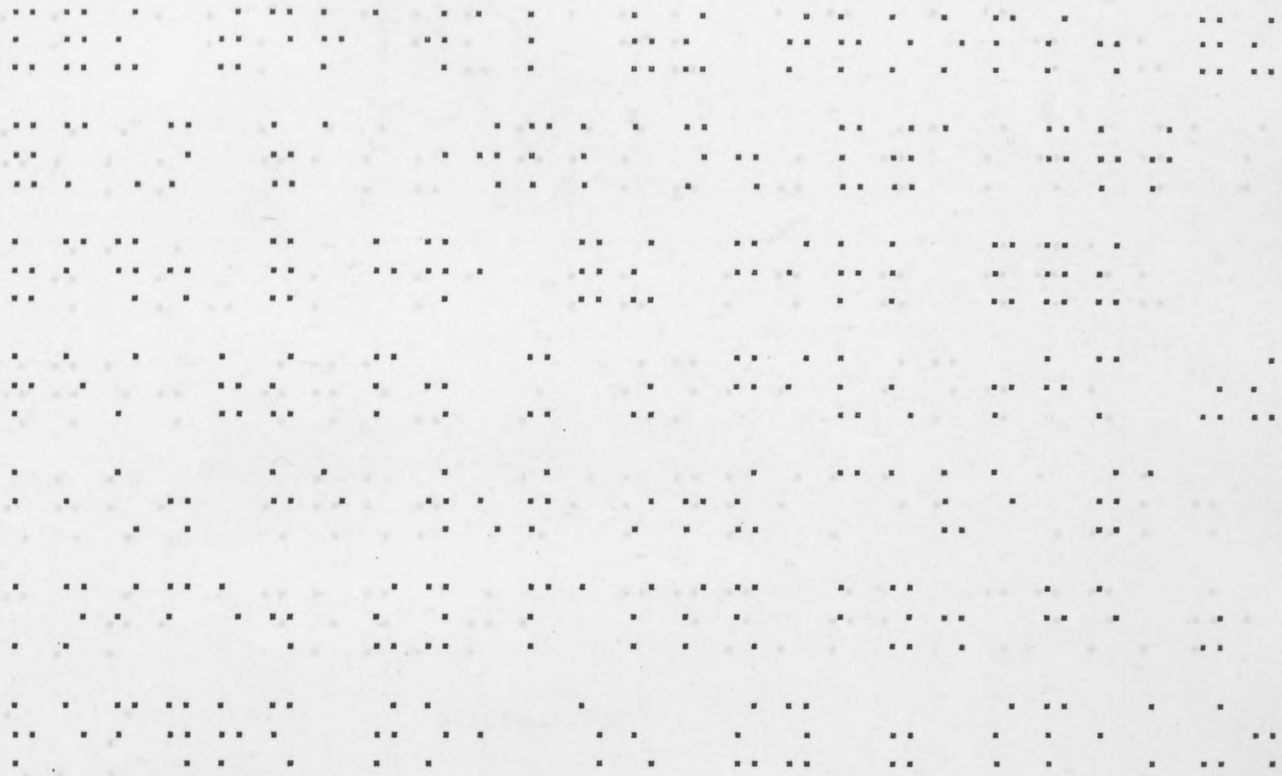


Figure A.7.2

The sentences for testing space contraction

Figure A.7.3

The tests for the translation of punctuation marks

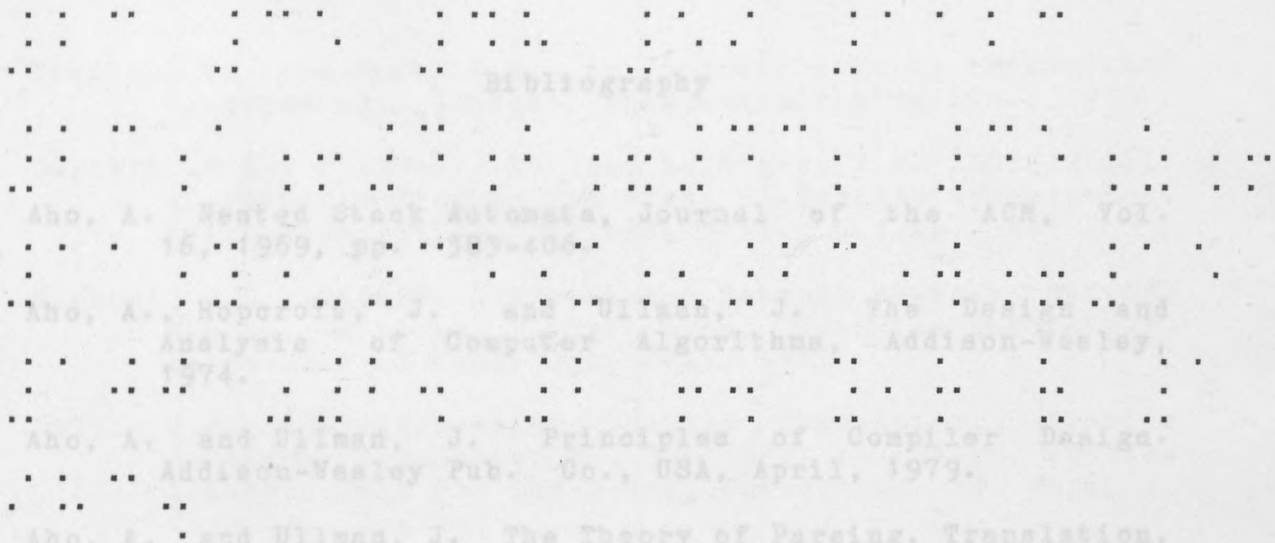


Figure A.7.4

The tests for the translation of numbers

Bibliography

Aho, A. Nested Stack Automata, Journal of the ACM, Vol. 16, 1969, pp. 383-406.

Aho, A., Hopcroft, J. and Ullman, J. The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

Aho, A. and Ullman, J. Principles of Compiler Design. Addison-Wesley Pub. Co., USA, April, 1979.

Aho, A. and Ullman, J. The Theory of Parsing, Translation, and Compiling., Vol. I : Parsing, Vol. II : Compiling. New Jersey: Englewood Cliffs, 1972.

Atkinson, M.P. Network Modelling, Ph.D. Thesis, University of Cambridge, May, 1974.

Bobrow, D.G. and Fraser, J.B. An Augmented State Transition Network Analysis Procedure. Proceedings of the International Joint Conference on Artificial Intelligence. Washington, D.C., 1969, pp. 557-567.

Chomsky, N. Aspects of the Theory of Syntax, MIT Press, Cambridge, Mass., 1965.

Davidson, I. Braille Translation by Microcomputer. International Conference on Computerised Braille Production - Today and Tomorrow, London, May - June, 1979.

Day, N.G.P. Non-Linear Structures and ATNs, Thesis Proposal, Computer Laboratory, University of Cambridge, January, 1978.

Day, N.G.P. Correct Editing with ATNs. IUCC Bulletin, Vol. 1, No. 2, Summer, 1979.

Douce, L.J. Some Developments in Computer-Aided Information for the Blind. Proc. of IEEE, Vol. 123, No. 1, January, 1976.

Earley, J. An Efficient Context-Free Parsing Algorithm. Ph. D. Thesis, Dept. of Computer Science, Carnegie-Melon University, Pittsburgh, Pa., 1968.

Earley, J. An Efficient Context-Free Parsing Algorithm. Communications of the ACM, Vol. 13, No. 2, February, 1970, pp. 94-102.

Etherton, M. Data Structures for a Network Design System, Computer Journal, Vol. 14, No. 4, 1971, pp. 366-374.

Findlay, W. and Watt, D.A. An Introduction to Methodical Programming. London: Pitman Publishing Ltd., 1979.

Gerhart, W.R., Millen, J.K. and Sullivan, J.E. DOTSYS III: a Portable Program for Grade II Braille Translation. Mitre Corporation, Bedford, Mass., U.S.A., 1971.

Gibbons, J.J. TINGES--A Transition Network Grammar Editing System, Diploma Dissertation, Computer Laboratory, University of Cambridge, July, 1977.

Gill, J.M. Microprocessor Braille Translator. Braille Research Newsletter No. 10, 1979, pp. 25-28, Warwick Research Unit for the Blind, University of Warwick, Coventry, England.

Gill, J.M. and Clarke, L.L. Braille Research Newsletters (No. 1-11). Warwick Research Unit for the Blind, University of Warwick, Coventry, England. July, 1976 to July, 1980.

Gill, J.M. and Humphreys, J.B. An Analysis of Braille Contractions. Braille Research Newsletter No. 7. Warwick Research Unit for the Blind, University of Warwick, Coventry, England.

Grimes, J.E. Network Grammars.

Haynes, R.L. and Siems, J.R. Computer Translation of Grade II Braille. International Conference on Computerised Braille Production - Today and Tomorrow, London, England, May 30th - June 1st, 1979.

Hopcroft, J. and Ullman, J. Formal Languages and their Relation to Automata, Addison-Wesley, 1969.

Jensen, K. and Wirth, N. Pascal: User Manual and Report. New York: Springer-Verlag, 1978.

Kaiser, G.E. Automatic Extension of an ATN Knowledge Base. Communications of the ACM, Vol. 24, No. 9, September, 1981.

Kaplan, R.M. Augmented Transition Networks as Psychological Models of Sentence Comprehension. Second International Conference on Artificial Intelligence, British Computer Society, London, 1981.

Knuth, D.E. The Art of Computer Programming: Sorting and Searching. Massachusetts: Addison-Wesley Pub. Co., 1973.

Lawes, W.F. The Feasibility Study into the Usage of Computers by and for the Blind. London, R.N.I.B., 1975.

Linden, C.A. Grammars for the Control of Structured Data. Ph. D. Thesis, Cambridge University Computing Lab., 1978.

Lomet, D.B. A Formalization of Transition Diagram Systems, Journal of the ACM, Vol. 20, 1973, pp.235-257.

McGillivray, Robert. Aids and Appliances for the Blind. Newton MA 02158: The Carroll Center for the Blind, issue No. 11, Winter, 1984.

Perrault, C.R. Augmented Transition Networks and their Relation to Tree Manipulation Systems. Ph. D. Thesis, University of Michigan, Dept. of Computer and Communication Sciences, Michigan, USA, 1975.

Peters, P.S. and Ritchie, R.W. On the Generative Power of Transformational Grammars, Information Sciences, Vol. 6, 1973, pp.49-83.

Riseman, E.M. and Ehrich, R.W. Contextual Word Recognition Using Binary Digrams. IEEE Transactions on Computers, Vol. C-20, No. 4, April, 1971.

Riseman, E.M. and Hanson, A.R. A Contextual Post-processing System for Error Correction Using Binary Digrams. IEEE Transactions on Computers, Vol. C-23, No. 5, May, 1974.

Ritchie, G. Augmented Transition Networks and Semantic Processing. CSR-20-78, Dept. of Computer Science, University of Edinburgh, Edinburgh, Scotland, January, 1978.

Rounds, W.C. Context Free Grammars on Trees, Proceedings of the Second ACM Symposium on Theory of Computing, 1970.

Rounds, W.C. Complexity of Recognition in Intermediate Level Languages, Proceedings of 14th IEEE Symposium on Switching and Automata Theory, 1971.

Royal National Institute for the Blind (R.N.I.B.). Braille Primer with Exercises. London: R.N.I.B., 1969.

R.N.I.B. International Conference on Computerised Braille Production - Today and Tommorrow. London: May-June, 1979.

R.N.I.B. Standard English Braille: Grades I and II. London: British National Uniform Type Committee, 1971.

Rubinstein, R. and Feldman, J. A controller for Braille Terminal. Communications of the ACM, Vol. 15, No. 9, September, 1972.

Shannon, C.E. Prediction and Entropy of Printed English. Bell System Tech. Journal, 1951, Vol. 30, pp. 51-64.

Silverman, B.S. A Microcomputer-Based Braille Converter. Braille Research Newsletter No. 11, August, 1980, pp. 4-11.

Slagle, J.R. Artificial Intelligence: The Heuristic Programming Approach. New York: McGraw-Hill, 1971.

Ullman, J.R. A Binary n-gram Technique for Automatic Correction of Substitution, Deletion, Insertion, and Reversal Errors in Words. Computer Journal, Vol. 20, No. 2., November, 1975.

Wirth, N. Algorithms + Data Structures = Programs. Prentice Hall, Inc., Englewood Cliffs, New Jersey, USA, 1976.

Woods, W.A. Augmented Transition Networks for Natural Language Analysis. Rep. CS-1, Computing Laboratory, Harvard University, Cambridge, Mass., 1969.

Woods, W.A. Transition Network Grammars for Natural Language Analysis. Communications of the ACM, Vol. 13, No. 10, October, 1970.

Woods, W.A. and Kaplan, R.M. Lunar Sciences Natural Language Information System, Final Report, Bolt, Beranek and Newman, Inc., Report No. 2378, 1972.