Text and Braille Computer Translation

A dissertation submitted to the University of Manchester Institute of Science and Technology for the degree of Master of Science, 2001.

Alasdair King
Department of Computation

28 September 2001

Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Acknowledgements

I gratefully acknowledge the great support of my supervisor, Dr Gareth Evans, without whom this dissertation would not have been possible.

Abstract

This project is concerned with the translation of text to and from Braille code by a number of Java programs. It builds on an existing translation system that combines a finite state machine with left and right context matching and a set of translation rules. This allows the translation of different languages and different grades of Braille contraction, and both text-to-Braille and Braille-to-text. An existing implementation in C, allowing the translation of languages based on 256-character extended-ANSI sets, has been successfully integrated into a Microsoft Word-based translation system.

In this project, three Java implementations of this translation system were developed. `LanguageInteger` is a port of the existing C code. `Language256` uses the same translation language files but is coded using Java programming idioms. `LanguageUnicode` is based upon the use of Unicode to encode characters. Each implements a `Language` Java interface, which defines common public methods for the classes in accordance with object-oriented software development principles of encapsulation and reuse.

All the implementations performed translation correctly on a range of different operating systems and machines, demonstrating that they are platform-independent. `LanguageUnicode` was able to use language data files obtained over HTTP from a webserver. The implementations performed well relative to the native C program on a high-specification machine, but their performance was strongly dependent on available system resources. `LanguageInteger` performed fastest, and more consistently, across the range of platforms tested and is suitable to serve as a component in future development as a part of a platform-independent translation application. `Language256` did not perform as fast, so its development should be discontinued. `LanguageUnicode` performed least well, suffering from using Java Strings to represent language information. It should be recoded using arrays of ints. This can be based on the `Language256` program, which uses this representation internally. It is recommended that Java Beans be developed from the classes to facilitate future development with them as applets, GUI components and network applications.

The three classes are supplemented by two more Java programs for creating the language rules tables used by the programs, a test language and test input that will allow the validation of future implementations of Language, and full documentation of the classes in the standard Sun API format for future development.

Contents

1 Introduction

Braille is a system of writing that uses patterns of raised dots to inscribe characters on paper. It therefore allows visually-impaired people to read and write using touch instead of vision [rni2001]. It is a way for blind people to participate in a literate culture. First developed in the nineteenth century, Braille has become the pre-eminent tactile alphabet. Its characters are six-dot cells, two wide by three tall. Any of the dots may be raised, giving 2^6 or 64 possible characters. Although Braille cells are used world-wide, the meaning of each of the 64 cells depends on the language that they are being used to depict. Different languages have their own Braille codes, mapping the alphabets, numbers and punctuation symbols to Braille cells according to need. Braille characters can also be used to represent whole words or groups of letters. These *contractions* allow fewer Braille cells to encode more text (dur1996), saving the expensive printing costs of Braille text and making Braille faster to use for some experienced users (wes2001, lor1996b).

Modern computer translation of Braille is of benefit to Braille users. Scanners allow printed documents to be transformed into accessible computer documents. This text can then be translated into Braille for output to a Braille printer or directed to special Braille output devices. Braille input devices like the Perkins Brailler are designed to allow Braille code to be entered directly, and Braille users can successfully use a standard QWERTY keyboard. Braille code is stored on computers as North American Computer Braille Code, a mapping of the six-dot Braille code to ASCII English text character values (kei2000).

Braille translation is not a trivial task, however, because of the need to correctly perform the contractions. They complicate translation logic and introduce many idiomatic rules and exceptions. For example, the contraction for “OF” in American Braille cannot be used unless it is applied to letters pronounced the same way as the word “of”. This means that “OFten” can be contracted but not “prOFessor” [bra1999]. Despite these complications, translation programs have been developed, based on dictionaries of correct translations or complex rule systems. Few remain in the public domain for examination, however. One translation system still public is a translation system developed at UMIST [ble1995, ble1997]. This combines a large set of rules, relating input and output text, with a finite state machine that allows the application of rules to be controlled by comparing left and right contexts. This system translates input text using a set of character rules, determining what characters are valid for the language and their attributes, a finite state machine decision table, and a set of translation rules containing wildcards for matching input text. These parts together constitute a complete *language rules table*. Contraction and different types of translation can all be supported within the same language rules table because of the state table. Idiomatic translations can be supported by the translation rules and the context matching. This system is very flexible and can be used to translate to or from Braille code any language for which a language rules table can be created.

An implementation of this system has been produced at UMIST, the C program *BrailleTrans* [ble1995, ble1997]. It works with ANSI 256-character sets, used to represent characters on most computer systems (fow1997). BrailleTrans does not contain any language information internally. The language rules table containing all the information for the language being used is loaded from a machine-format file when BrailleTrans is first executed. BrailleTrans can use any language rules table interchangeably. The language rules tables can be created in simple plain text files by non-technical users. They are then converted into the machine format by a second program, *Mk*. BrailleTrans can therefore translate any language for which a language rules table file has been created. So far, Standard British English, Welsh and prototype Hungarian have been developed, both contracted and uncontracted and text to Braille and back. BrailleTrans has also been used to add Braille translation functionality to the popular Microsoft Word word processor [ble2001]. This integration provides a friendly and familiar interface to the translation system for users.

BrailleTrans is fast and efficient, but it has limitations. It runs only on 32-bit Microsoft operating systems. It handles only 256-character sets, which do not allow the encoding of non-Western characters and do not supply a unique value for every different character. This makes a context-free encoding of characters impossible - there is no single correlation of characters to unique values, so text cannot be translated without first knowing what language it written in. These limitations are addressed in this project by the development of a number of Java implementations of the UMIST translation system. Java, as described in Chapter 3, allows for platform-independence - Java programs can run on any system with a Java interpreter, the Java Virtual Machine (JVM). JVMs are widespread, although the range of Java library classes a JVM supports can vary so not all Java library classes may be available on every machine. It also supports the use of 16-bit Unicode characters, which escapes the restriction on supporting only languages that use a 256-character set. The Java implementations developed adhere to the Java 1.1 standard, making them compatible with the majority of JVMs. Java also allows for the use of object-oriented programming approaches, which aim to promote re-use and minimise errors.

Chapter 4 describes how three Java programs, or *classes*, were developed to perform translation. Two have the same functionality of the existing BrailleTrans program and use the same 256-character set language files. One, *LanguageInteger*, is a port of the existing C BrailleTrans program to Java, for comparison of output and to provide a highly-optimised benchmark. The other, *Language256* is a more object-oriented class that uses more Java programming techniques. These Java and object-oriented techniques are intended to make implementation easier and the resulting class simpler to maintain. The third Java program, *LanguageUnicode*, uses Unicode characters to represent text. All three were designed to implement a Java *interface* for Language that defines the public methods and variables that must be provided by the program. This is intended to establish the implementations as objects and facilitate their use as components in future translation applications. Two Java replacements of the existing Mk program convert the language rules tables from their plain text to machine formats - one handles the legacy 256-character files used by LanguageInteger, Language256 and BrailleTrans, the other the Unicode-based files used by LanguageUnicode. Both improve on Mk by allowing the text files to contain escape characters that code for other characters, allowing any Unicode or 256-character set to be represented in a strictly ASCII text file. The separate human-edited language rules table files were maintained to allow easy production and editing of languages. A number of command-line translation utilities demonstrated the ease of using the classes and their common interface for translation within a larger application.

The results of the implementations are described in Chapter 5. A test translation language and input and output files confirmed that the classes did translate as required on a variety of operating systems. The speed of translation differed between the classes, and was related to their different designs. Analysis of the execution of each of the classes was performed to identify possible coding improvements, but few such improvements were found to be obvious. All of the classes are documented internally in sourcecode and externally using the Sun Javadoc system to produce comprehensive documentation for subsequent development with the classes to form larger applications. A number of applications of the classes were investigated.

The project was therefore successful in producing platform-independent, component Java implementations of the UMIST translation system, including a Unicode language translator. All of the classes conform to good object-oriented design and with their documentation are well suited to future development as parts of a larger translation application. Chapter 6 provides more detail on these conclusions and specific proposals for future work on and with the classes.

2 Current state of Braille and computer use

2.1 The Braille code

Blind people cannot use printed or displayed text. They need a tactile or audile means to read and write. Braille was created in the nineteenth century to fulfil this need [rni2001]. Characters in Braille consist not of visual symbols, printed or displayed, but of physical symbols constructed of raised dots on paper. It is a system of reading and writing based on touch rather than vision, where characters are *embossed* rather than ink printed.

Each Braille character, or *cell*, is composed of a rectangular array of six positions, each of which may be filled by a raised dot. They are numbered:

```
1 • • 4
2 • • 5
3 • • 6
```

This provides 2^6 or 64 possible characters. The cell with no dots at all, an empty space, is used to separate words and for layout, just as in visual text. This leaves 63 cells for characters. This allows a simple one-to-one mapping of any given common Indo-European alphabet to Braille cells. Letters, numbers and punctuation can be recorded in Braille by single cells or very simple combinations of them. Translating text to Braille is then a trivial process of converting written text character by character. Some examples from British English Braille [rni1992]:

Character	Braille equivalent
J	⠚
!	⠋
7	Number symbol (⠼) and 7 symbol (⠛), ⠼⠛
3	Number symbol (⠼) and 3 symbol (⠉), ⠼⠉
T	⠞
H	⠓
E	⠑
THE	⠞⠓⠑

However, Braille languages do not use only these simple mappings. They also use abbreviations, or *contractions*, where single Braille cells or combinations of cells represent whole words or sequences of letters. These contractions fall into a number of categories. A common word can be indicated by its initial letter alone, called a *wordsign* (e.g. letter 'B', Braille cell '⠃ ', represents "BUT") or by another unique cell (e.g. '⠯' for "AND" and '⠿' for "FOR"). Some wordsigns can be used within longer words to replace some of the letters of the longer word (e.g. "bAND" or "FORce"). Some characters are used solely within other words to indicate particular common groups of letters - called *groupsigns* (e.g. "CH" used in "CHap"). (All examples again from Standard British English [rni1992]).

These contractions allow text to be written with fewer Braille cells than would be the case if each visual character were rendered by one Braille cell. Contracted Braille requires around 20% fewer characters than uncontracted Braille [dur1996]. Braille cells must be larger than text characters, since they must be recognised by touch, and embossing Braille cells is more expensive than printing ink characters. Therefore, contraction both reduces the final size of Braille texts, making them easier to store, distribute and use, and saves money in printing costs. It has been a part of Braille since the nineteenth century [irw1955].

The use of contractions, however, also makes Braille translation much more complex. It requires the introduction of many more translation rules. These are required to handle the greatly increased number of possible translations for a given piece of text. To give some examples, again from British English: the phrase "per cent" is represented by the Braille cells "⠱⠏ " (normally the colon and letter 'P' cells). However, the phrase "per diem" does not have a similar contraction, so the word "per" cannot be translated until the word after it is identified. Similarly, "have" is a wordsign, so when it appears alone it can be contracted, in this case to '⠓ ' (letter 'H'). However, where it forms part of a word, it cannot, so "haven" or "shave" must not use the wordsign contraction. Groupsigns, however, can be used as part of words, but some can only be used in a particular position in the word - for example, the contraction for "less" can only be used at the end of a word, so "less" and "lesson" will not use the contraction but "bLESS" will. Some contractions can be used where the original meaning of the letters is maintained but not otherwise, so "upon" can be contracted in "whereUPON" but not in "coUPON". Many more examples could be given. They all have one common result, however, which is to require a larger number of more complex rules - general contraction rules, exceptions, special cases, preferred contractions – than would be required if simple uncontracted Braille code were used. Even without contractions, Braille would have to handle formatting, italics, text abbreviations (like "NATO" or "St. John") and other complications. The contractions, however, are what turn Braille translation from a trivial to a more complex task.

There is a non-economic reason for contracting Braille, which is perhaps more important since improved printing facilities have reduced the cost of embossing Braille. Contracted Braille may be faster for Braille users to read. Fewer characters mean less time needs to be spent in navigating the text, for example in moving to the beginning of the next line of text, which can take 6-7% of reading time [lor1996b]. If words are recognised in their entirety rather than character-by-character then contraction will speed recognition of contracted common simple words [wes2001]. However, errors in recognising words also slow down reading speeds [lor1996b]. Because of the complexity of Braille contraction, errors (by the producers of the text or by the reader) are more likely to occur in reading and writing contracted Braille. Contraction will help reading speeds in skilled readers using accurate and consistent texts, but may hinder less skilled workers or those using inaccurate texts.

For this reason, some Braille users dislike the use of some of the less common contractions, alleging that they make Braille harder to learn by requiring more rules to be learnt with little benefit in space saved. This can mean that writers in Braille do not observe even official contractions. (It may be that computer Braille translation leads to more standardised Braille within languages. However, rare printed Braille materials are giving way to electronic documents in those countries rich enough to afford computer translation, so computer translation may in fact allow local custom to be maintained, translating from a common non-Braille text resource).

2.1.1 Types of Braille

Braille translation is further complicated by the existence of many different Braille codes. Six-dot cell Braille is the most frequently used tactile writing system. (An eight-dot variant is not as widespread). Even languages that might use hundreds of characters for written text will use a phonetic alphabet version of the language to allow it to be translated into Braille code. For example, Chinese Braille uses a version of the 58-character phonetic *pin-yin* Chinese alphabet rather than the thousands of characters in the traditional *wen-yan* alphabet [zeh1999]. Because there are many human languages, and they have different alphabets and their own mapping of characters to Braille cells, there cannot be therefore a single interpretation of a single six-dot Braille cell. For example, the Braille cell '⠮' in British English is the groupsign contraction for "the"; in Welsh is "dd"; and in Russian is the letter "yery" [rni1992]. Therefore, any given Braille cell can mean different things depending on the language represented and its character set. Translation requires the language to be identified or assumed before it can be performed. A human translator can identify a language relatively easily, or at least identify which language a language is not. A computer translator needs to be informed or assume the language to use.

There are similarities between many Braille codes, reflecting attempts to standardise the codes or in many cases their natural evolution from the original French Braille code. For example, European languages use the same Braille cells for similar letters, alphabets permitting. Braille cell '⠁ ' encodes letter 'A' in English, 'α' in Greek, "ah" in Russian and "A" in Welsh [rni1992]. American, Australian and British English Braille code are very similar because of deliberate attempts over decades to standardise them and thus be able to share Braille materials [lor1996a]. However, even amongst these three closely-related nations, every one using English, the Braille codes differ in practice even if they do not now in theory [bra1999]. Different Braille traditions affect the Braille code used by translators and preferred by readers. For example, American Braille has tended to use fewer contractions. British Braille tends to allow the contraction of vowels even when the pronunciation is not the same (for example, in both "idEA" and "milEAge") while American Braille forbids it [ble1995]. These differences persist, even where contractions have been officially standardised across the two languages.

A further complication is that Braille documents are laid out in a standard format - titles are indented by a certain amount, lines are of a certain width, a defined distance is left between lines and so on [rni1992]. This is designed to allow Braille users swiftly to navigate Braille documents and books. These layout rules again differ between languages.

The official standards established in English include uncontracted, or Grade I Braille code, and contracted Grade II Braille code. Attempts have been made, largely successfully, to ensure that Grade II is standard in both American and British Braille. Other contraction grades exist, such as Grade 1.5 and Grade 3, to name but two [lor1996a]. These reflect historical stages of development of the modern codes and national and regional variations.

Finally, Braille codes have been devised for non-literary purposes - mathematical and scientific work [dur1991], music [brl1997], and for specific specialized purposes like chess playing and knitting patterns [sul1997]. These will again depend on a specific set of rules for that language or notation, with their own idiomatic semantics and structure, and require their own translation rules.

2.1.2 Computer Braille

American Computer Braille, also known as North American Braille Computer Code (NACBC), or Universal Computer Braille Code, has become the *de facto* international standard in recording Braille characters in computer files, much as ASCII has become the standard computer text [dur1991]. It maps the 64 Braille six dot cells to the ASCII set of characters. NACBC files will be displayed as Braille text when viewed with an appropriate Braille viewer or fed to a Braille printer.

The Braille characters making up an NACBC file will mean different things according to the Braille language used, just as a printed Braille book could be in English or Korean while still consisting of six dot Braille. This means that, like Braille in general, it is not tied to English but can store any language converted into six-dot Braille. It can be assumed that NACBC is sufficient for encoding six-dot cell Braille text in any language.

NACBC values are given in Appendix 1.

2.2 Use of Braille with computer technology

The recent spread of personal computers has brought new benefits to Braille users that have access to them. Computerised Braille translation is one of these benefits, but it is not a straightforward process because of the complexities of Braille contraction.

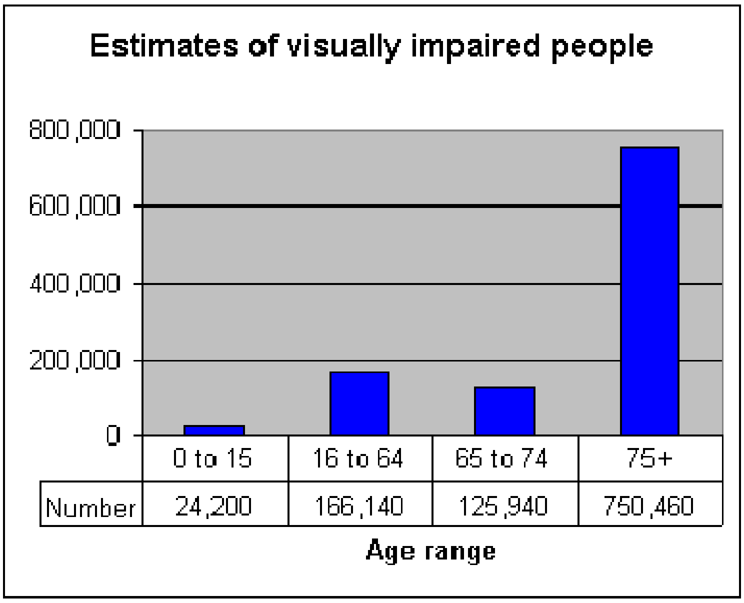
2.2.1 Difficulties of computer Braille translation

Contraction makes Braille translation difficult. Without it, Braille translation would be a relatively simple task, mapping text characters to Braille cells. Each language (English, French) or code (chess, maths) would need its own unique mapping dictionary, but the operation would be trivial. Contraction greatly increases complexity, so computer translation is generally difficult. For example, in German, there are many words that are formed from the concatenation of other words, similar to the English "uphill". Many potential Braille contractions will arise from the new juxtaposition of characters at the joins. However, these must not be contracted under German Braille rules, which respect the separate origin of the constituent words. Correctly identifying German Braille contractions therefore requires either etymological information or a comprehensive German dictionary of exact translations [obe1990].

This demonstrates how contraction makes it impossible to define a small set of simple, general rules that perfectly translate text into contracted Braille code. Instead, rules are needed to define when particular letters are mapped to particular Braille codes. To continue the American Braille code example given in 2.1.1, an American Braille translator should distinguish between the use of "EA" in "idEA" and "EA" in "milEAge" because they are pronounced differently. The translator will have to maintain a dictionary of words containing "EA" to ensure that they are all translated correctly, because there is no regular pronunciation rule for "EA" in English and therefore no simple translation rule can be devised. The translator can save dictionary space by using a general rule defining what to do with an occurrence of "EA" for which it does not have an entry. This will require a smaller dictionary, but at the cost of being less comprehensive adnt therefore less accurate. Greater accuracy in contraction can only be obtained by an increased number of language-specific rules or dictionary entries. The many idiomatic exceptions and special cases for contraction must be handled individually for complete accuracy. This is not solely a problem of differing pronunciation. Even the translation of a language with very regular phonetic pronunciation like Czech [hed1998] would require an understanding of the punctuation, abbreviation and significance of layout in text. Computerised Braille translation is not a simple case of requiring faster computers or larger and more detailed dictionaries, but developing techniques that address the more subtle and difficult task of producing accurate translation to the standard needed by Braille users for effective use of the translated Braille text. The possible benefits of developing such techniques are discussed in the next section.

2.2.1 Justification of computer Braille translation

Braille has come under attack in recent years, with studies showing decreasing usage due to changing patterns of education [bru1991]. Alternative technologies, like speech synthesis, now exist for many of its application. It can be safely assumed that sighted people will use ink-printed texts, rather than Braille. The potential Braille population therefore consists of the visually impaired, partially sighted or blind. The UK Royal National Institute for the Blind provide estimates for visually impaired people in the UK population [rni2000a]:



There is a large visually impaired population in the UK, but the great majority have become so because of the effects of age [rni2000b]. Cataracts, age-related macular degeneration and diabetes are the leading causes of sight loss, and the incidence of each increases with age. The visually impaired are therefore largely older people. Older people may find it more difficult to master the technical skill of reading Braille than children taught it from an early age. Teaching older people

Braille will also give less return than teaching the young, who will gain more years of use from the knowledge. Limited resources will tend to be distributed towards young learners.

There are still young people with visual impairment. These may be from genetic factors, accidents and other factors. However, modern social practice has mediated against raising children with visual impairments separately from sighted children where possible [bru1991]. 96% of those classified as blind have some vision, and practice now is to use better teaching materials, specially-designed resources and modern technology to allow them to be part of sighted society rather than part of blind society. This is a change in the practice of previous decades [sch1999]. In addition, many children who have a visual impairment also have a learning disability, which will make it harder for them to learn Braille [mca1996].

It would appear then that Braille is an old technology that has been superseded by modern technology and social changes. This is not the case. There are fundamental reasons for continuing with Braille:

Reading and writing Braille code is a form of literacy, Literacy is an vital component of modern knowledge and society, and Braille code permits literacy for blind people. Using audio-only technology denies literacy to Braille users.

- **Braille code is silent.** Alternative technologies to Braille may not be usable in all circumstances. A speech synthesizer and speech interpreter forbid complete privacy and intrude on the local environment. This would not be appropriate in, for example, an office environment.
- **Braille code is accurate in reading.** Speech synthesis, an alternative to Braille, may introduce indecipherable meanings. Problems will arise from words not known to the synthesizer, mis-spelt words (e.g. “the fihs swam” is clearly a typographical error when read, but “the fizz swam” is not so clearly identified and corrected when heard), or with words pronounced differently according to context (e.g. “The book was not read because I wanted to read it later”). Reading straight from the text removes a potential source of error.
- **Braille code allows interpretation of the text by the reader, not by another.** Any reader who wishes to interpret a text themselves - an actor, or a reader for pleasure - may want to form their own interpretation of a text, not be forced into that of a disinterested computer program or particular actor. Inaccuracies in interpretation will be possible.
- **Braille code is cheaper than computer technology.** This is a simplification. The mechanism to print and reproduce Braille is expensive, and the volumes produced are

large and costly to transport and store. However, when printed, Braille documents are fully accessible to Braille readers without electricity, computer hardware or any other expensive equipment. This will be of great advantage in areas where the advanced technology is not available or prohibitively expensive. Speech processing, output or input, is resource-hungry and requires powerful, new and expensive machines. Technology to allow computer Braille translation may be affordable. Technology to replace Braille entirely may not. Braille is a more accessible technology, reflecting its 19th century roots.

(It can be argued that Braille code will always be necessary for situations where technology is unavailable or inappropriate, such as note-taking, but this is not be an argument for producing a computer Braille and text translation service and so is not advanced here)

For these reasons Braille use should still be a significant part of blind written culture. Despite the difficulties in computerising Braille translation, the potential rewards justify attempts to produce Braille computer translation services.

2.2.3 Potential Braille translation services and users

Braille users will not simply perform text and Braille translation but combine this capability with input and output devices suitable for their needs. Computerised Braille translation offers a number of different services:

Translating non-Braille text for output to a Braille production device

Personal computers with powerful text manipulation and output capabilities allow for the production of simple printed Braille documents more efficiently, just as non-Braille word processing has aided ink document production. Specialised Braille printers or embossers can produce Braille more quickly than was possible in the past, although they are still expensive and relatively slow compared to ink printers. They can be fed straight from a computer output with North American Computer Braille Code. This allows non-Braille users to produce Braille as well as saving Braille users translation time. It will also be of service to those requiring large-scale Braille document production. For example, UK utility companies are now required provide bills in Braille (hms2000) and will want to automate their production to reduce costs. More recent technology affords new methods of accessing text. Braille text can be output to specialised Braille displays or *liners*. These use an electrical mechanism to project up pins onto the user's fingertips. This simulates the passage of physical Braille characters under the fingers of the user, and is faster than a speech synthesizer. It also may be more appropriate, in an office for example. It is a potential user of real-time text to Braille translation.

Translating input from a Braille device into non-Braille text

Braille keyboards, like the Perkins Brailler, permit Braille users to enter Braille directly into a computer. They use special keyboards to allow Braille characters to be entered quickly and accurately. Other key-Braille mappings are possible for standard QWERTY keyboards and other devices, such as a mobile telephone keypad. Real-time translation in particular would lend itself to using these Braille inputs to produce non-Braille text directly on the computer. An obvious example is the production using Braille of a non-Braille document for non-Braille users - say an email or word processed document. Other applications might be a Braille-aware input for a mobile telephone or personal computing device. This might allow text messaging, simple note-taking or control of the device's functions.

Intermediate translation for libraries

This covers the creation of libraries of Braille documents or non-Braille documents from the opposite source. Common references, or records of events, or works of fiction, can be translated and stored. This allows future access without further translation. A simple set-up of a computer with a Braille printer would benefit from a stored library of commonly-produced works, which could be called up and printed out on demand. Translation in this case would probably be followed up by manual checking of the translated document, as befits its more formal existence as a permanent work of reference. This greater care is analogous to the level of proof-reading afforded an ink-printed text, depending on its desired status - a temporary electronic mail where low quality is tolerated compared to a high quality permanent reference text, for instance.

Adapting to changes to Braille code

A computer Braille translator will be ignorant of old codes. Changes to the Braille codes used can be implemented immediately, without retraining or error. Braille has changed repeatedly over time [irw1955].

A translation program that fulfils some of these possible functions will need to meet some functional specifications. It will need a high level of accuracy, in contracted and uncontracted Braille, to be useful. It will need to work sufficiently fast for its possible uses, some of which require real-time performance. It will need to be maintained relatively easily to allow for changes in Braille rules and changes in use. Finally, the Braille market is a small one, divided amongst users of many different languages, and a translation program that was flexible enough to translate a wide range of languages and grades of contraction is more likely to be adopted and justify its development.

2.2.4 Actual computer Braille translation services in existence

Despite the difficulties inherent in developing computer Braille translators, these programs do already exist. Large-scale translation is used to produce Braille versions of textbooks and other large works for sale. Smaller systems are used by teachers and individuals for teaching or personal use.

The market-leader in commercial translation products is produced by the Duxbury company in the USA [dux2001]. They provide many functions, including translating both word-processed documents and plain text, integration with other applications, bulk translation and graphical interfaces. There are versions for individual users, integrated with the Microsoft Windows and Apple Macintosh desktop operating systems, as well as large-scale volume translators. They are feature-rich, constantly updated but relatively expensive. Some fourteen languages are supported.

Another system designed for desktop computers is WinBraille. This is produced by the Index Braille company [win2001], which manufactures embossers and thus produces a free text-to-Braille product to encourage Braille use. This supports 18 languages in a WYSIWYG word-processor style, allowing Braille code to be edited directly. It is designed to produce output for the Index embosser. It uses a rule-based translation system, and the rules can be edited by users for local variations. It also supports grades of translation, straight character or abbreviation.

The HotBraille Internet website [hot2001] provides a service, free to US citizens, that allows the entry of a message on a website that will be translated into Grade II American Braille and posted free of charge to a US address. This is intended to encourage non-Braille users to regard Braille as a valid communication medium and to enter correspondence with Braille users.

2.3 Approaches to performing Braille translation with computers

With the profitable commercialisation of several public translators the number of approaches available for study in the public realm has decreased. Approaches have tended to be based around the use of dictionaries of specific translations and limited rule systems because of the trade-off between translation rule numbers and accuracy described in Chapter 2.2.1 [ble1997]. The rules work with a ‘window’ of the input text that can potentially be translated, and examine the context to the right of the window to see whether a rule is correct (to determine whether the window is in the middle or end of a word, for example). A finite state machine [sul1982, wer1982] determines whether the translation is performed. This may be still the basis for commercial products, but they have ceased to be published in the public domain.

Finite state machines involve the complications of state and control tables and many rules. An alternative approach using only the matching of left and right contexts of the translation window was developed [sla1990] with the intent of producing a system easy to develop and maintain by non-experts, such as those interested in developing Braille translation for different languages but lacking computer programming skills.

More recent approaches have used state tables that perform left and right context checking, limited dictionary definitions for common but particularly irrational translations, and simple finite state machines [das1995]. This compromise allows the flexibility of a state machine to be combined with a simpler rules and dictionaries that are easier for non-technical people to use.

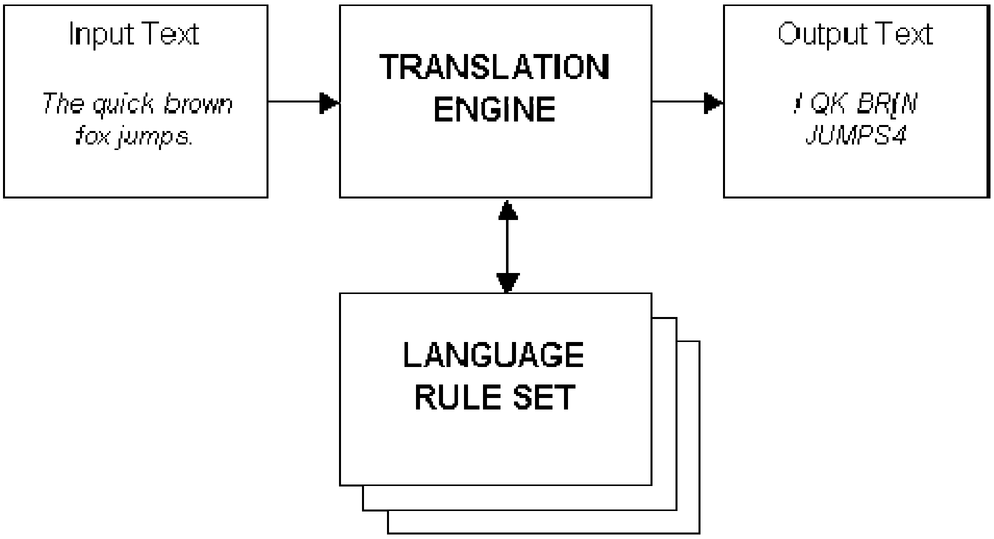
UMIST has developed a system using this approach [ble1997]. This uses a finite state machine in a strictly limited fashion, primarily to handle grades of Braille within the same language and to allow a single set of rules to double as translation to and from Braille for a language, and a simple (if lengthy) list of character translation rules that can be edited directly by non-technical users. It hopes therefore to be easily adapted for translating text into contracted Braille for many languages, such as Hindi, where commercial products are unavailable or too expensive. The further development of this system will form the basis of this project, so the next chapter describes it in detail.

2.4 The UMIST translation system

The UMIST translation system has been implemented and is a working Braille and text translator. It is detailed in the papers by Paul Blenkhorn (1995, 1997) [ble1995, ble1997] from which the following detailed description is taken. The system will form the basis of the development performed in this project, so a full understanding of its function is necessary. It is one of the few published pieces of work on text and Braille translation in recent years.

2.4.1 Translation mechanism summary

The UMIST translation system is structured thus [ble1995, ble1997]:



A finite state machine, the **translation engine**, works with, but is independent of, a single **language rules table**. The former contains the translation algorithms and functions, and the latter all of the translation information for translation of one language in one direction (e.g. English to Standard English Braille). The translation engine can use any language rules table, so any language can be translated to or from Braille code if the language rules table is constructed. The language rules table consists of a set of **translation rules** and a **decision table**.

During translation the engine works along the input text, character by character. It attempts to match a window of input text starting with the current character with one of the translation rules in the language rules table. A successful match with a translation rule must match a segment of text, the context – the text to the left and right of the window - and the state of the engine. The engine state is controlled by a finite state machine, using the contents of the decision table, and regulates which subset of the language translation rules can be used. This allows both contracted and non-contracted translation can be supported from the same language rules table. The translation rule then provides the translation for that window of input text, which is appended to the growing output text, and the engine moves along the input text to the next unmatched character.

A description of the structure of the language rules table will clarify the translation mechanism.

2.4.2 Language rules table

A language rules table contains all the information for translating a given language to or from Braille, so each language supported has at least one table. The individual components of a table are:

1. Character mapping rules.
2. The decision table.
3. The specification of the translation rule wildcards.
4. The translation rules proper.

2.4.2.1 Character mapping rules

These specify:

1. The transformation, if any, that should be applied to each character of the input data to normalise it for translation. They can map, for instance, lower-case to upper-case letters, or types of brackets ('{', '[' or '(') in the input to a single bracket character. This has the effect of reducing variation in the input text and requiring fewer translation rules for the language. They could also be used for appropriate mapping of control or layout codes, or particular character relationships.
2. A number of flags indicating the characteristics of the character, e.g. whether text, punctuation, a number. These are used when determining whether a translation rule matches the input text context.

2.4.2.2 The decision table

This table controls the operation of the finite state machine. This provides more flexibility, but the very simple state machine does not overly complicate the system. It is typically concerned with switching between grades of contraction (such as Grade 1 or 2 in British English Braille code) and with managing the appearance of single letters, for example in postcodes, that must not be contracted (for example, a postcode “M1 1CH” must not have the “CH” contraction applied to it).

The table can be represented as a simple matrix of boolean values, machine states by input classes. For any given translation rule the current machine state is cross-indexed with the rule's input class (see next section) to obtain a true/false result. This determines whether the rule can be applied if the contexts and focus all match.

2.4.2.3 Translation rules

The translation rules provide both the output for the translation system and the criteria that govern whether a rule can be applied. They double as both simple character-to-character mapping rules and more detailed dictionary definitions. Each rule consists of six components:

Input class

A number used by the finite state machine to determine whether this rule can be used in the current machine state.

Focus

The string of characters that must be matched with the window of characters from the input text. With a given character at the start of the current window of input text, the engine will examine the group of translation rules with foci starting with this same initial character. When a match is found the engine moves along the input text to a new input text segment with a new first character, which will in turn determine a new focus group. “AND”, “AN”, and “A” might be some of the members of the focus group with initial character “A”.

Left and right context

When matching a window of the input text against the translation rules, consideration is given to the surrounding text or context. This is the core mechanism that allows this system to perform complex translations and cope with idiomatic constructions. It allows idiomatic text to be translated correctly - for instance, in the text "St. John" the full stop does not indicate a sentence end and should not be translated with the Braille codes for a sentence end. Matching text to the left and right of the focus allows these circumstances to be accommodated. The context that must be matched can be defined as a sequence of specific characters or use language-defined wildcards.

Output

This is simply the text that results from the successful matching of a translation rule. It forms the output from the system. It will consist of North American Computer Braille Code characters in text-to-Braille, or (unformatted) translated characters from the correct language character set for Braille-to-text. (Though there is nothing in theory stopping it being used in another translation task, the current use is for Braille and text translation).

New state

The application of the rule may result in a change of machine state. This provides the new state, if any.

The translation rules form the dictionary of the language rules table file, and provide information for the context matching and decision table control. They detail the idiomatic rules and particular exceptions, as well as the default translations for single characters. The more translation rules provided, the more accurate the language rules table will be. An important feature is that the rules can be edited by non-technical users, since they are laid out in clear text. This allows new rules be added or old ones amended as necessary. A Braille current-affairs newspaper might want to add a specific rule to ensure the translation of the name of a prominent politician, for example. More generally, of course, the creation of a new set of translation rules is the main task in creating a new language rules table for a whole new language. Their simple layout is key to allowing non-technical users to perform this important task.

Within each translation rule focus group, the rules must be in inverse order of length of focus. The foci of rules that match may have subsets that also match. If the subsets are then checked first they will be matched, but this is an error since the longest possible focus should be matched where possible (or as the Braille rules put it, the contraction that saves the most space should be used). An example from British English: if the input string "`THEMSELVES`" translates to Braille code output "`!MVS`" but input string "`THE`" translates to "`!`" then the rule containing the focus "`THE`" must come *after* that containing "`THEMSELVES`" or the shorter focus will always be matched before the longer is reached. "`THE`" would always become "`!`" and the system would then try to match the remainder of the input text, in this case "`MSELVES`". This remainder does not independently resolve to "`MVS`". The translation will have failed to match the correct (best) contraction. The order of focus groups of translation rules is not important, since the translation engine will either jump to the start of the correct focus group anyway, or else start at the first translation rule and work through, which means no match can be made until the first rule of the correct focus group is reached.

2.4.2.4 Context wildcards

The left and right contexts of the translation rule can either match sim'ple literal characters or wildcards. Wildcards are symbols in the translation rules that indicate that a particular sequence of characters is required. These can match "types" of characters - punctuation, letters, numbers – and particular numbers of them. These types are defined by their flags in the character translation table. Each wild card thus has the following characteristics:

1. The wildcard character.
2. How many input characters the wildcard can match (zero-or-more, one, or one-or-more).
3. The character characteristic flags to match (e.g. "punctuation" or "whitespace"). These relate to the flags of the character mapping rules for individual characters.

An example to clarify: the English to Standard English Braille table specifies the wildcard '`;`'. It represents 0 or more "letter" characters, indicated by the setting of the "letter" character flag. This may then match any sequence of characters defined in the character mapping rules that are all similarly possessed of a set "letter" character flag, for example 'A', 'B', 'C' and so on.

2.4.3 The translation algorithm and process

The details of the language rules table provided in the preceding sections should allow the algorithm of the translation engine to be understood (again from Blenkhorn [ble1995, ble1997]):

```
initialise and load language translation table
translate input text into normalised form using character
    mapping rules
current_character = first character of input
while current_character <> end of input do
  begin
  start at first rule whose focus begins with current character
  match = FALSE
  do
    if       focus matches
         and state is ok
         and right context matches
         and left context matches
      begin
      add output from matching rule to output buffer
      set new state according to matching rule
      match = TRUE
      curent_character moved along input by length of focus
      end
    else
      go to next rule
  until match
  end
return output buffer containing translated text
```

Before translation starts, the input text is normalised, ensuring that any characters in the text are appropriate for that language. This can remove upper and lower case, for instance. It can simplify translation and can also allow more flexibility if required.

At any one time the engine tracks:

1. where translation has got to in the input text, `current_character`
2. the machine `state`
3. the current rule under examination

It iterates through the translation rules until a match is found for all the conditions, obtains the output and moves along the input text to the next unmatched character. The algorithm is high-level, but simple.

There is no explicit support for capitalisation (or any other text formatting) within the design. This might seem surprising, since capital letters may have different meanings. However, this might not be best supported by handling capitalisation separately for the same reasons that preclude generalised translation rules, the importance of context and idiom. Capitalisation, if required in a language, must be handled in the translation rules - adding more rules to differentiate between capital and lower case use.

The design does not make explicit what should output when no match is found for an input character. A space, or the input character, or another character, or nothing may be written, at the discretion of the implementer. The choice will reflect the final user requirements for an implementation of the system.

2.5 Implementing the UMIST translation system: the BrailleTrans program

This translation system has been implemented at UMIST by Paul Blenkhorn [ble1995, ble1997]. A C program, BrailleTrans, implements the translation engine, and a number of language rule tables have been devised to be used by it. The technical challenges of implementing the translation system are addressed successfully be the implementation, so a close study of its mechanism is of great interest for any future development of the translation system. The BrailleTrans source code, the way it was built into a working program, and the language rules tables it uses were all studied in detail, and are described in the following sections.

2.5.1 BrailleTrans implementation

BrailleTrans is coded in the C programming language. It is designed to be compiled and run on one of the Microsoft Windows 32-bit operating systems (Windows '95/'98/Me or Windows NT4/2000). When compiled to an executable program, in native code, it resides on a machine as a Windows Dynamic Linked Library (DLL). This is a binary executable that supplies a public interface to the Windows operating system and can thus be utilised by other Windows applications with comparative ease. It is designed to be fast and resource-efficient.

BrailleTrans uses the primitive `int` or `char` data types to store characters in the input text, output text and within the language structures. They can and are used interchangeably, since C, and most older text formats, assumes a 256-character set, where one 8-bit byte encodes one character. While a human user might regard the system as working on letters and symbols, the system is in fact working with integers in the range 0 to 255. It is up to the user to view the input text, output text, and most importantly the language files in the correct character set mode, and to operate the program in an environment where the program and applications will use the correct set in their dealings with BrailleTrans.

2.5.2 Language translation rule table

The language rules tables are loaded in from disk, where they must be local to the program. Each language rules table is stored in one separate file. These files are in a compressed machine-readable format, not a plain text format. The language files are created initially with a standard text editor in a plain text format, with supporting comments and in a human-friendly form. Another C program, Mk, is then used to compress these files into the machine format that is used by BrailleTrans. Mk removes the comments and compresses the language rules table by removing all formatting and layout. This makes loading the files much faster, since less parsing of the data file is required, and reduces the size of the language files, but still allows non-technical users to manage and create the language rules tables in the simplest format possible. For Standard British Braille, the relative file sizes are 30.4 kilobytes and 22.0 kilobytes for the uncompressed and compressed files respectively.

When loaded into memory only the character mapping rules and information on translation rule focus groups are stored in a dedicated C data structure. The translation rules, wildcard rules and decision table rules are stored as simple `char` arrays. A pointer to the translation rules array is used to read and navigate the translation rules. This allows fast and low-level access to the data, but has required the writing of low-level algorithms in BrailleTrans rather than the use of any common C libraries working on higher-level data structures.

2.5.3 Loading translation rules tables

The structure of the translation rules table files is important, because it reflects the structures used to represent the language information in memory and therefore the operation of BrailleTrans. Each of the data structures making up a translation rules table is examined below, in the order they are recorded in the language rules table machine-format file.

2.5.3.1 Version numbers

BrailleTrans checks first four bytes of the language file for the following integers, in order: 0, 7, 12, 8. The first is the version number, labelled as such in the plain-text version of the language file. The latter three are undocumented version numbers that are written into the machine-format file after the version number by Mk. They are then checked by BrailleTrans on loading the file. If they are not present, translation is halted and an error code indicating an invalid language rules table file is returned. This allows older invalid language files to be identified and newer correct versions obtained. Without the version numbers, the only indication that the language rules table was out-of-date might be from erroneous translation or program failure.

2.5.3.2 Character translation rules

The 256 character translation rules, corresponding to the 256 characters in the 256-character set assumed to be the basis for text encoding, each take three bytes. The first byte is the code for the character to be used - a mapping character for the normalisation process. The second is intended to form part of a system for capitalisation, but is not used in this implementation. The third byte contains the eight type flags for the character in the form of an eight-bit register. For example, the British English translation rules table has seven types - text, wildcard, punctuation, capital, whitespace, roman character and digit - and one unused type. A character may be in several categories. Characters may have more than one flag set – the wildcards used are also punctuation characters in the British English translation rules table, so have both flags set.

2.5.3.3 Wildcard flags

This table for the wildcards consists of one byte holding the number of wildcards, followed by that number of triplets of bytes, each making up one wildcard entry. The three bytes making up a wildcard entry consist of:

1. The wildcard character, again an 8-bit byte.
2. The number of characters the wildcard should match, whether zero or more, one exactly, or one or more. These are expressed as integer constants in the program for readability and consistency.
3. The flags that a wildcard character must match, corresponding to the character type flags described above. These are used to compare with the flags of input text characters being matched against a left or right context containing the wildcard.

Like the decision table and translation rules the wildcards are held in a one-dimensional character array, of size `(number of wildcards) * 3`. If a character in a left or right translation rule context is encountered with a set wildcard flag, the wildcard table array is iterated through three at a time until all the wildcard characters have been examined to find a match. Since the number of wildcards is part of the language rules table the array can be iterated through with a `for` loop. If a match is found, the loop is terminated and the process of matching the wildcard flags against the correct number of input text characters is initiated. If not, the character is compared directly to the input text as a non-wildcard character.

2.5.3.4 States, input classes and decision table

The file next holds one 8-bit integer for the number of states, S, in the finite state machine for this language, another for the number of input classes, C, and then the full decision table. This consists of S bytes, each a row of the table, repeated C times to make up the full table. Any non-zero entry indicates that an attempt to match the machine state s_i against a translation rule input class c_j succeeds, and a zero entry indicates a failure. The array is thus be indexed for the state and input class with the index `(i – 1) * S + (j – 1)`. (In BrailleTrans, this value is used directly as the argument to a conditional operator, since C regards a zero result of any expression as `false` and non-zero results as `true`. This is an efficient way to code).

2.5.3.5 Translation rules

The remainder of the file is taken up with the translation rules. These are loaded directly into another `char` array. The number of rules and their total size is not recorded in the language file, so the character array is declared with a capacity of 25000 items. If this is exceeded when loading the table, translation is halted an error code returned.

Each rule is of variable length, and is a sequence of bytes:

- One byte, the input class for the rule.
- One byte, the total number of bytes in the rule.
- A variable number of bytes holding the rule contents, just as in the unedited file: left context, '[' character, focus, ']' character, right context, '=' character, output.
- The number zero, indicating the end of the rule contents.
- The new state after the rule is applied, or zero if no change of state is required.

The array is accessed via a global pointer to the array, which is fast and powerful. It is difficult to code with without error, but a normal part of C programming. Navigation from one rule to the next rule may therefore be accomplished by reading the input class and size of rule, and then using the size to increase the value of the pointer, so that it then points to the input class of the *next* rule. The first rule is coded to be of input class zero and size one. The final rule is followed by the character '#', so an input class of '#' for a rule indicates the end of the translation rules.

If only linear navigation within the rules were permitted the program would be slowed by having to iterate through the translation rules from the first rule for every input character. The translation rules are however arranged in their focus groups. This permits navigation directly to the first rule of the focus group when a matching input character is encountered. This is accomplished by the use of a hash, part of the character rule data structure described in the next section.

2.5.3.6 Character rule data structure

A C data structure, called `ch_info`, is used to hold the character mappings and the hash index to the translation rules. The latter is used to jump straight to the correct set of translation rules whenever a new segment of input text is to be processed.

`ch_info` consists of an array of C `struct`s, each containing the character to be mapped to in normalisation, the character in upper case, and the character flags. These values are obtained straight from the character rules table. Each `struct` also contains as an `int` the index in the translation rule array of the first rule of the matching focus group, called the *hash*. The hash to the translation rule array is populated as the translation rules are read into the translation rule array. Each new rule is checked to see if the hash value for the character that begins the focus of the current rule has an value in the `ch_info` array, and if not, it takes the index of the current rule being loaded. This then allows a quick hash into the translation rule array when translating, rather than requiring a search through from the first rule.

There are 256 entries in the array, one for each member of the 256-character set being used. Again, BrailleTrans deals with characters as numbers in the range 0-255, corresponding to the integer values of these character sets. Therefore, a character can simply be used as an index to the ch_info array. The character rule for the letter ‘A’ is therefore `ch_info(‘A’).` This is one of the benefits of limiting BrailleTrans to 256-character sets. It is convenient and used throughout the program.

2.5.4 Translation engine

This section describes how BrailleTrans uses the language rules tables to perform translation.

2.5.4.1 DLL structure

As a DLL BrailleTrans possesses a number of functions that are not related to the process of translation but allow the program to be integrated into the Windows operating system. This is intended to provide it with the ability to be accessed and managed by the OS in the most efficient manner. These access functions will not be discussed further, since they are specific to this platform and implementation.

There are two functions that allow the program to be called from other applications running on the machine to perform translation.

`int InitBrl (char *name)`

This is passed the filename of the language rules table file to use, loads the table from the file on disk, and parses the contents into the data structures for translation. Since the filename is passed as a pointer, the DLL can be passed a full system path and filename for the language file. The integer value returned indicates success (zero) or an error (a number of informative error codes are defined).

`int Translate(unsigned char *input_txt, unsigned char *output_txt)`

This is passed the input text (or rather a pointer to it in memory) and the destination for the output (again, a pointer to it). Completion returns zero to indicate success.

The program therefore progresses in the following order:

1. A call to `InitBrl` with the path and filename of the language file.
2. A call to the `Translate` function with the input text.
3. The translation engine translates the input text using the language rules, writing the output text to memory using the output pointer. This part of the operation is completely written in standard C, without Windows variations. This maximises potential portability of the code to other platforms.

The translated text is not passed back to the calling application, but has been amended in memory by BrailleTrans. The calling application passed the output pointer to BrailleTrans when it initiated translation, and thus can use the same pointer value to access the translated text.

2.5.4.2 Normalising the input text

On calling `Translate` BrailleTrans performs the initial normalisation stage, where the program simply converts the entire input data by substituting each character with the character from the character mapping rules indexed by its integer character value:

```
int i = 0;
while (input_txt[i] > 0) {
    input_txt[i] = ch_info[input_txt[i]].input_trans;
```

```
                i++;
        }
```

2.5.4.3 Commencing translation and the main translation loop

`Translate` now in turn calls `convert`, again with pointers to the input and output data memory. `convert` now performs the main translation routine. This involves keeping track of where in the input text the translation has reached, and calling `find_match` with this location and the pointers to the input and output text. `find_match` returns the number of characters that have been successfully translated, and this is used to move along the input text until it is exhausted. If no match is found, `add_to_output` is called to simply copy the unmatched input character to the output text. This is the approach most likely to produce usable output, but makes it more difficult to diagnose errors.

2.5.4.4 Matching the input text

When `find_match` is called it finds the correct language translation rule for the input text and current machine state. The first character to be matched from the input text is used to obtain the hash value in the `ch_info` table, which provides the new value for the pointer to the translation rules array.

This rule, and each successive rule until a match is found or the focus category changes, is checked for a match. The following criteria must be met, in order:

1. Focus - an exact match of the rule focus and the input text, starting from the current input text index.
2. State - the decision table is consulted with the machine state and the rule input class to test if the rule can be applied.
3. Right context - a match between the input text and the right context of the rule, starting from the current input text index plus the length of the current rule focus and moving right along the input text and context.
4. Left context - match as the right context, but starting at the character to the left of the current input index and the right-most character of the left context and moving left along the input text and context.

The contexts allow the use of wildcards, already detailed above. These allow a single translation rule to translate many input text possibilities. This therefore requires the language to have far fewer translation rules in total. This makes the language easier to build and manage and the translation process faster, since fewer rules need to be matched and rejected for each input character, but does require the most complex code in the program. However, left and right context matching is performed by the same function, `wild_match`, which takes as its arguments the input text, where to start looking for a match, and also the direction to proceed with matching the input and rule context. This can thus match either the left or right contexts. This neatly reuses the most difficult part of the BrailleTrans code.

BrailleTrans, as an implementation of the UMIST translation sytem, makes no allowance for capitalisation in the input string - everything is converted to upper case for translation. There is therefore no way to control capitalisation except through the translation rules. There is a case for implementing a mechanism for translating capitalisation outside of the translation rules, if it is to be translated at all, since it would greatly increase the complexity of a language rules table (more rules could be introduced to cope for upper- or lower-case letters separately, or many case-insensitive wildcards used). This would be very complex.

The program is highly-optimised, using very efficient data structures and providing compiler information identifying the most efficient memory allocation of variables. For example, frequently-accessed variables are declared with the `register` compiler instruction. Wildcard matching is performed efficiently by comparing flags values using the AND operation. The expression `(input_char_flags & wildcard_flags) != 0` demonstrates the use of the '`&`' operator to perform the logical test "Does the character type match the wildcard type?" (i.e. the expression evaluates to `true` if this is the case). The input character flags are obtained from the `ch_info` structure and the wildcard flags from the wildcard table.

2.5.5 BrailleTrans implementation issues

This summarises the action of BrailleTrans. Without going into further detail, some other features of the BrailleTrans program are of interest:

- There is no checking of the boundaries of the input text array when comparisons with the contexts are performed. C permits access to array items outside of the array boundaries, but the results of accessing such areas of memory is quite unpredictable. This will usually work in practice, since the unpredictable values returned by comparison with areas outside the array are very unlikely to match the rule requirements, and will fail. However, it is not good programming practice.

- It is not possible to know before running the translation process what the final size of the output will be. Translating contracted Braille into text will produce more final output text than input text. Since the output text is held in a character array, it must be declared in advance. The ratio of the size of the final output text to the size of the input text must be assumed in advance of running translation. If too small a ratio is assumed, the output text array may be overflowed. This will only occur when contracted Braille code is being translated into text, and is likely only to be a risk when the Braille code fragment is very short and contains significant contraction and the output text is proportionally very large. This might happen when a document is parsed into sections and then translated, when for instance a short heading might be passed for translation. Conversely, if too large a ratio is assumed, memory resources are wasted. (BrailleTrans terminates the output text produced with a zero, so the end of translated text is clearly delimited. Beyond this zero each array member can have any value, since C does not initialise variables to defaults when they are declared).
- There is no way to control the state of the translation finite state machine - it is always set to 1 by the `Initialise` function, called before each array of input text is translated. In the British English language file this corresponds to Grade 1 uncontracted Braille.
- Because BrailleTrans requires a 256-character set, and hard codes this into the program operation is speedier and far simpler. However, the onus is on the user to operate BrailleTrans in the correct character set environment for the language and input text. BrailleTrans is ignorant of the difference between character sets. It will still produce a translation, whether meaningful or not, since it handles only the integer values of characters. For example, using a language rules set based on the Western European Latin1 character set with input text from the Eastern European Latin2 set will produce output text, but output text translated according to a foreign Braille code.

2.6 Using BrailleTrans: the Word translation system

BrailleTrans on its own is not useful to a Braille end-user. It needs to be embedded into a larger application that allows the end-user to perform Braille translation. This might be a standalone application, or part of another application. At UMIST BrailleTrans has been integrated into the Microsoft Word word-processor. The implementation is detailed in Blenkhorn and Evans, 2001 [ble2001]. It consists of:

1. The Microsoft Word word-processor, a Microsoft Word macro, and a Visual Basic program.

2. The translation engine, BrailleTrans.
3. A translation rules table for Standard British Braille.

Together this system allows the Word users to translate text into Braille code. An added Word menu item, "Convert to Braille", activates translation from within an open document. The option is presented as to whether to use Grade 1 or Grade 2 Braille. British English, Welsh and prototype Hungarian are supported. The translation system operates, and the translated document is produced as a file on the local drive, presented to the user as a new open document in Word.

The system runs on 32-bit Microsoft Windows operating systems (Windows '95, '98, Me, 2000, and NT 4) and versions of Word from Word '97 onwards. It is at an advanced commercial state. The BrailleTrans implementation of the translation system fits well into the complete system, providing the correct functionality at levels of performance in excess of those required by the other components of the application. This demonstrates the validity of the UMIST translation system in being used in a real context. A closer examination of the application is of benefit.

2.6.1 Word and Visual Basic wrapper

A Word document is not simply a flat plain text file. It structures the document according to the purpose of each part of it, header, section, basic paragraph and so on. It thus provides presentation context as well as raw content. This is called the Word Object Model [ble2001], and has direct parallels with the document model used in JavaScript and HTML [www1998]. It allows the properties and content of a document to be accessed simply and independently. When a document comes to be translated into Braille the document information is used to inform the translation layout and formatting of the finished Braille document, which complies with the Braille layout rules.

This high-level and Word-based parsing of the document is done by the Word macro (a simple scripting language program that can be associated with a document) and the Visual Basic program. The Word macro and VB program produce plain text for the BrailleTrans DLL, so text formatting is not a consideration. BrailleTrans is only required to translate plain text according to the grade and language required.

The Visual Basic routine that calls BrailleTrans assumes that the output text will be no larger than double the length of the input text, and declares the input and output arrays accordingly.

2.6.2 Rationale, use and users

Microsoft Word is a very common word processing package, and is very likely to be the normal word processor for any typical worker with Braille [seg2001]. Embedding the translation system within MS Word therefore makes it very much part of the familiar user experience, and links directly with the familiar way of producing ink documents. This is good Human-Computer Interaction (HCI) practice, and encourages use of the system.

Word also allows reading of many other electronic document formats, including ASCII text and HTML documents. Other formats can often be cut and pasted directly into Word, an action that again will be familiar to most users. This makes the system as usable as possible for any worker who wants to translate to or from North American Computer Braille Code.

2.7 Limitations of current implementation that can be addressed in this project

The current BrailleTrans implementation and Word system meet the needs of standalone office workers. They do not address some important issues of computer Braille translation. Having described them in detail, it is possible to identify some problems with the implementations. The attempted solution of these problems will form the basis of the development work of this project:

- **They are platform-dependent.** They require Microsoft Word and a 32-bit Microsoft Windows machine (Windows '95, '98, Me, 2000, NT 4). This is a very common combination [seg2001], so this is not too great a limitation, but it still prevents the use of the system on other environments, like Unix, Linux, Apple Macintosh or Solaris. It also limits the system to desktop machines. Small consumer devices (Personal Digital Assistants, mobile telephones) are not supported, and neither is it suitable for server use without further development.
- **The system must be fully installed on the local machine**. It is stand-alone, designed to be run on a machine with all the program components installed and present. This limits the machine to the translation languages that have been located by the machine's user and moved to the machine, local to the translation system files. An ability to be run remotely or accessed over a network would be desirable.
- **The use of 256-character sets is hard coded and immutable.** 256-character sets are the basis for most character encoding systems, but the increased use of computers by users with non-Western languages with larger or incompatible character sets threatens

to make the current implementation incompatible with the languages of many potential users. This is an issue with special relevance, since one of the stated aims of the BrailleTrans implementation was to produce a piece of software that could be easily adapted for other languages, for example Hindi [ble1997].

An investigation into the possible technologies and strategies that can be used to solve these problems is now required, and is the subject of the next chapter. The combination of limitations and potential solutions will allow a precise set of development requirements to be identified before coding begins.

3 Solutions to current implementation limitations and development requirements

This project will attempt to address the limitations of the current implementation. A number of important implementation decisions need to be taken to select the technology to be used. The first and most important is the choice of implementation platform.

3.1 Implementation platform and implications

The current implementation's key problems are platform dependency, restriction to local machines and 256-character set limitations. The first two of these stem from the language and technology used, not from the state machine/translation rules approach. If there *were* a demonstrable problem in this approach then using the UMIST system would be pointless, and if there was a demonstrable problem with the C implementation then the best solution might be to fix the C implementation. As it is, solving the current problems will involve implementing the UMIST translation system again in a new environment. The choice of new environment will determine the work needed, and the likely problems that will be encountered, which makes it a crucial first step.

3.1.1 Possible platforms and architectures

The key choice in implementation is the programming language to use. This appears to be a trivial problem, since all modern programming languages permit the implementation of any algorithm. However, a programming language is not simply a way of encoding algorithms. Every language is different in more fundamental ways - platform-dependent idiosyncrasies; a characteristic profile when running; access to low-level controls in the operating system, performance. Also, recent years have seen a move to a more software-engineering and object-oriented approach to software development. This aims to save time and money and reduce software errors. Software languages form a vital part of this move. First, they take difficult and error-prone tasks like memory management away from programmers. Second, they provide fully integrated tools for common programming tasks, for example networking, database access and GUI management.

There are numerous candidate languages, each with strengths and weaknesses. The main contenders:

- **Perl** [wal2001] is a fully interpreted language popular amongst the UNIX, Open Source and Web developer communities. It does provide object-orientation and platform portability. However, it is not common outside of the these communities, which restricts the pool of potential future developers of the system for UMIST, and the interpreter is not present on the majority of systems, which restricts the pool of potential users.
- **Microsoft Visual Basic** [mic2001] is very commonly used for simple application development. It is interpreted and provides a Graphical User Interface (GUI)-centred approach that allows the quick construction of simple applications. However, it is Microsoft platform-specific and lacks a good component architecture.
- **C#** [mic2000] is the programming language component of the Microsoft corporation's new technology strategy, and is network-focused and very similar to Sun's Java. It is designed around the Microsoft operating system, so again is platform-specific. It has not yet been widely adopted and is not yet widely supported.
- **Java** [sun1999a] is a Sun Microsystems language and applications environment that is platform-independent, handles international text and is widely used in all environments. Its Application Programming Interface (API) and Java library classes are extensive and powerful. It fulfils all of the requirements for improvements to the UMIST system, and will therefore be used for the project.

3.1.2 Java

Java is a programming language and operating environment devised by Sun Microsystems [gos2000]. Based on the oak programming language, it is object-oriented, high-level and general-purpose with strong Internet inflences, Its syntax is deliberately very close to C and C++, for familiarity for programmers moving to Java, and it is marketed as the successor to C++ by Sun.

Java is partly a compiled and partly an interpreted language. Java source code is not interpreted directly but is compiled into compact Java byte-code. This does not execute as machine code, but is interpreted by a Java Virtual Machine (JVM). A JVM is a program compiled to native code, typically written in a low-level language like C. It insulates the Java byte-code from the platform on which it is running. This makes the byte-code platform independent. The JVM manages all system-specific tasks like memory management and file handling. It also allows the language to be standardised for any platform. For example, the primitive integer `int` type in Java can be defined as always being a 32-bit signed integer, while a C programmer must be aware that the size and "signedness" of the `int` type can vary from platform to platform and must code accordingly. Going one step further, Java compiled byte-code can be trusted to run on any JVM on any platform, and to a reasonable extent this is true. Conversely, any platform can run any Java program so long as a JVM is written for that platform. This insulates Java programs from concerns about operating systems or platforms. The same Java-based webserver, for instance, should run the same way on a Solaris, Unix, Linux or Windows NT machine.

Java is strongly object-oriented. Java uses *classes*, which define types of objects. Every standalone Java program is an instance of a Java class with a special `main` method. While the most primitive data types, integers and characters are not objects, arrays and strings are. Every other Java data structure must be an object. Library functions are accessed through class or `static` methods. All this object-orientation encourages good software engineering approaches to problems and time-saving component re-use.

Classes are organised into pseudo-hierarchical *packages*. For example, the classes `DataInputStream` and `File` both perform input and output functions, and are both found in the `java.io` package. However, the `java.util.zip` package is not in itself found in the `java.util` package, although the nomenclature of the package names would suggest that it is. Different package names in fact indicate entirely separate sets of classes. The Java API comes with many such "built-in" packages that provide tools for performing common programming tasks. The most basic `java.lang` package contains the core Java logical operators, primitive data types and object architecture, and is the basis for every subsequent Java class. API packages are located in the `java.*` and `javax.*` hierarchies and provide classes for GUIs, networking, mathematical functions, abstract data types, input and output and many other tasks. These packages are similar to the C libraries that are used by C developers for tasks like common string operations, but they are more fundamental to the language. They are available to developers and programs (depending on the version of Java used, as described below). They can also be regarded by developers as being fully optimised for performance, and trusted to be error free (or at least to have well-documented bugs with known workarounds). They also allow compiled Java classes to be much smaller than standard executable programs, since the libraries functions reside in the JVM, not the Java program - useful if the Java class is transmitted over a network before use.

Java also manages much that would be the programmer's responsibility in other languages, for example automated memory allocation and reclamation, explicit casting, error handling, and array bounds checking. Taking responsibility for these functions away from the developer prevents the developer from making errors with them. Sun intends this to make the language easier to program with and more robust in use. Finally, Java lacks some features that are important features of programming in other languages but which also are the most common sources of errors. There are therefore no pointers, no `goto` statement, no operator overloading or multiple inheritance in Java. All these factors combine to make Java an easy language in which to construct working programs, although the strict object-orientation makes it conceptually more complex than a simple procedural language like C.

The object-oriented nature of Java has made it very modular and flexible. In addition, many of the Java packages provide good tools for network operations. Java environments are therefore able to load Java components and access resources from remote network sources. This architecture has been very popular for Internet applications, notably in Java *applets*, Java classes that are embedded into standard World-Wide-Web Hypertext Markup Language (HTML) webpages. A Java-enabled web browser, on encountering the page, will load the Java class from over the network and utilise a local JVM to run the class. This provides functionality to the static HTML webpage – the webpage can now offer dynamic, variable content, like a game or graphics application, all within the applet, without requiring a return to the server for a response to every user action. The JVM verifies the class before running it to ensure that it is consistent with the Sun-defined applet security policy, so this is a relatively safe method of running foreign code on a local machine. Applets cannot access the local file system, for example, and no Java class can perform low-level activities like writing directly to memory, so the potential for malicious applets is severely limited. Applets never heralded the birth of operating-system-independent distributed network computing that Sun hoped for, where local desktop machines would simply download applications as needed. However, they did provide functionality to webpages just as the Internet was becoming popular in the mass market, and therefore both Microsoft Internet Explorer (version 3 and higher) and Netscape Navigator (version 2 and higher) provide the ability to use applets. This means that nearly every modern desktop computer is able to run Java applets - a considerable user base. The same modularity that lead to Java applets has helped Java to win a place in a wider range of computing uses, including low-level consumer devices like personal digital assistants (PDAs) and mobile phones, and high-level distributed applications and enterprise servers. This makes Java a good choice for producing a component like an implementation of the UMIST translation service. It maximises the number and range of potential users of the implementation.

These features have made Java popular, recently threatening to overtake C/C++ in popularity [gal2001]. This is a reason in itself to use the language. Support for using the it will be easy to come by, future work outside UMIST will be easier to arrange, and corporate and programming popularity guarantees a longer lifespan of support for any application. More specifically, Java now forms a key component of undergraduate and postgraduate taught courses in UMIST. This has provided the author with skills in Java, and will have biased the author towards the use of Java in this project. Any future development building on this project will likely be performed in-house by future students, who will in turn be most proficient in Java. This is a strong argument for using Java for any UMIST student who wishes to leave work of some benefit.

3.1.3 Different Javas

Java is popular in different areas of computing, which can be broadly split into three - small consumer devices, desktop applications, and large enterprise server applications. Each has different needs, though each is attracted to Java for much the same reasons. To address these different areas, Sun has allowed Java to fragment from the single standard of a few years ago. In addition, the computing industry has changed over time, and Java has changed to adapt. Java is therefore not a unified entity, but varies with age and supported environment. Deciding on a Java to use will define the Java library classes available and what potential user base exists, so must be decided before coding a new UMIST translation system.

3.1.3.1 Different Java versions

The core Java language has not changed since its introduction in syntax, primitive data types, object typing and other fundamentals. What has changed over time is the number of Java packages provided in the Java class libraries. These Sun Java library packages are included in every distribution of a JVM or Java software development kit (SDK). The size of the library - the number of packages, the number of classes in each package, and the number of functions or *methods* available in each class - has increased over time. This means that a Java program written using only Java version 1.0 classes will run fine on any JVM, but a Java program using, say, Java version 1.2 classes will require a JVM with the version 1.2 classes available. This has obvious implications for the portability of a given Java class.

The lower a version of Java used, the more common will be the JVMs to run the program. One of the issues to be addressed is the use of the system on a low-specification machine, so an older and less resource-intensive code version that avoids more recent and resource-hungry library class functions may be advantageous. Older Java library methods have been widely-used by developers as part of Java for longer, and therefore any bugs or performance issues are more likely to have been resolved by Sun or other JVM vendors in their JVMs. A lower version will also be more easily transferred to consumer devices, which commonly utilise subsets of standard Java. However, lower versions of Java contain some classes and methods that are deprecated; their use in the real world has indicated that they are flawed in operation or design, and although they still remain in the Java libraries for legacy support of older programs, Sun recommends that their newer replacements in the libraries are used instead.

Java is now in version 1.3, termed Java 2 by Sun, but this disguises some more complicated changes to the Java language described fully in the next section. For "standard" Java, the versions are 1.0, 1.1, 1.2, 1.3 and 1.4. The change from 1.0 to 1.1 heralded some important improvements to the language. Java Beans, an extension of the applet concept for enterprise-level applications, were introduced, and might be useful for a distributed implementation; Java Archive (JAR) files allow Java classes and resources to be compressed into single smaller files for easy and fast distribution over a network; object serialization allows Java objects to be written straight to disk or passed over a network, very useful for storing data structures like language tables. Perhaps most importantly, Java 1.1 introduced classes that handle input and output of Unicode characters. The new library classes and improvements of Java 1.2 to 1.4 are of less definite relevance to this project. The key changes were in the approach to producing GUI components with Java, where the new Swing design replaced much of the previous Abstract Windowing Toolkit (AWT). This is not relevant to the implementation of the translation system, which will produce a small component intended for use in a larger application. Java 1.1 provides vital library classes but Javas 1.2 to 1.4 do not. Java 1.1 will therefore be used in the project, with the caveat that any Java 1.1 classes deprecated in later versions are avoided.

On platforms where system resources are plentiful – modern desktop computers or dedicated servers for example – a later JVM will perform better than an earlier JVM. The additional resources, especially memory, are put to better use and performance can be markedly improved compared to older, ostensibly more efficient, JVMs. If using a later version of the library classes were necessary to access these improvements there might be a case for using a later version. However, a Java 1.1 class can, of course, execute without problems on a later Java 1.3 JVM. There is no restriction on a JVM except that it is *at least* Java 1.1 capable. A later JVM can be used where this is of benefit.

3.1.3.2 Different Java supported environments

Sun attempted to market Java, originally designed for embedded applications, into the desktop application market. It has not however been able to supplant platform-specific native applications in the Microsoft-dominated desktop market for performance and political reasons. The market has broadened both upwards, to enterprise-level servers, and downwards, to small consumer devices like PDAs and mobile telephones. Java as originally constituted was not best placed to capture these markets. Sun therefore accepted the commercial imperative of attempting to gain a share of these markets and split Java into different APIs and environments.

The current official split of Java comprises three segments, each addressing a different set of requirements [sun2000, sun 2001]:

Enterprise Edition (J2EE)

This is designed for servers, especially internet servers and database interfaces. It uses a component modular architecture of Java classes and Sun technologies like Remote Method Invocation and Java Database Connectivity. These provide functionality for connecting networked machines and resources.

Standard Edition (J2SE)

This is standard Java, closest to the original big Java market, desktop devices. It concentrates on applets and, to a much lesser extent, standalone Java applications.

Micro Edition (J2ME)

This is designed for small consumer devices with limited resources, unable to run JVMs that can support the full array of Sun Java library classes introduced in the recent Java versions (sun2001a).

This is a simplification of evolution of Java on consumer devices. Other Java-based systems from Sun, including EmbeddedJava (sun2001c) and PersonalJava (sun2001d), existed independently before J2ME, and a number of JVMs and consumer devices use these alternative technologies. Some consumer devices use J2SE, but specify an older Java version than the current (and thus limit the Sun library classes that are required to be supported). Even J2ME itself is not a definite subset of the Java language but a mechanism for hardware vendors to define an official subset for their devices, a *configuration* in Sun jargon, that can then be used by developers for writing Java applications for that device.

3.1.3.3 The implications of the Java version chosen

Java 1.1 is also supported by the Java Virtual Machine shipped in versions of the Microsoft Windows operating system, both consumer and enterprise, from Windows '95 to Windows 2000. Microsoft operating systems have the vast majority of the desktop operating system market, and a significant proportion of the server market (sha2001). (Sun also controls a significant proportion of the server market, and obviously includes a JVM on their Solaris server platforms). Due to legal conflicts between Sun and Microsoft no more advanced version of Java is provided by default on the Windows operating systems, though of course another JVM can be installed. Using 1.1 Java would allow Microsoft users to run the system without needing to install and configure another JVM from another vendor (for example, the Sun JVM for Windows '98 is an eight-megabyte download from the Sun website (sun2001e)). To give another example, it may be decided to implement the translation system in an applet to produce a web-based translation system. Again, the great majority of users use the Microsoft Internet Explorer browser, which runs Java 1.1. Java 1.1 therefore offers the best balance of functionality with potential user base.

In Sun's recent classification, this corresponds to a J2SE implementation. All J2EE environments will run the application: A J2ME environment may or may not, depending on the environment, but using only the packages of Java 1.1 will minimise any work required to transfer the program to a specific J2ME system.

3.1.4 Java issues

Java has not achieved its hoped-for pre-eminence in the consumer application market. One of its main non-political problems has been performance. Developers accustomed to the speed of compiled native-code programs written in C or C++ have been unimpressed with the speed of the object-oriented, high-level, partially-interpreted language [shi2000, eck2001]. The features that make it a powerful software engineering tool retard performance - heap memory usage, pointers for all objects, garbage collection. Taking difficult tasks, like memory management, away from developers may make it easier for them to write programs, but it also relies upon automated mechanisms to manage and optimise these tasks. This introduces a management overhead for the JVM, which must calculate the best way to assign and reassign resources. In addition, resource allocation is dynamic and must perforce allow leeway in resource assignment, where a human programmer might be able to identify the optimum distribution of resources. Skilled developers are denied the ability to bring different, more efficient approaches to these tasks by being insulated from the low-level operating system by the JVM.

Java's benefits can hamper its performance in other ways. Some Java library classes provide very convenient abstract data types (ADTs) for performing routine programming functions. For example, the `java.util.Vector` class provides a growable array, where the size need not be known in advance. This is very useful for a programmer. However, it is modelled internally by a normal Java array. When it reaches its capacity, adding another item forces the JVM to create another array of double the size, copy the old information over, and eventually perform garbage collection on the old `Vector` (hag2000). This operation becomes increasingly expensive as the `Vector` grows. This class must therefore be used with extreme care if the `Vector` will be required to expand considerably or if the likely deployment platform machine is of a low specification. The initial size set for the `Vector` becomes very important, weighing up memory wastage from choosing too big a `Vector` against the possibility of repeated internal array expansion from choosing a `Vector` too small.

Another problem is with object-orientation (sos1999). Object creation requires a string of calls up the object hierarchy and the construction of a complex Java object. Whenever a Java class is loaded by the JVM, most commonly when an instance of that class is created, the byte-code must be verified by the JVM before the new object instance is created. Both these factors militate against using objects in Java, which is difficult since they are so integral to the language. Other problems result from the features that give Java its robustness. For example, Java strings are objects and also immutable, so amending a `String` object requires creation of a new object (gos2000). The strong typing that prevents many errors in Java requires explicit casts, which are resource-hungry. Dynamic selection of methods at runtime is very flexible and good object-orientation but imposes a large overhead. Finally - a trivial example in many cases but important in applets - the JVM requires time to start up if not already in operation whenever a Java class is run.

Compilers and JVMs have improved over time and system resources have increased, but the translation classes must be designed and implemented with performance as a key consideration. There are many techniques for improving code efficiency. Metrics have demonstrated that in most programs where speed is critical most of the time taken is spent executing only a small, repetitive portion of the code. The proportion varies, but 90% of the time on 10% of the code is frequently cited as an accurate estimate (bel1997). This means that this small amount of performance-critical code can be optimised without sacrificing good design or practice in the bulk of the program. For example, declaring classes with `final` prevents subclassing and run-time dynamic method over-riding and speeds up execution, but is not best object-oriented practice. It would be acceptable if used very sparingly and only where absolutely necessary. It is strongly recommended within the programming community that coding for performance be avoided where possible, since this can lead to obfuscated code, reduced portability, the risk of new bugs, and the investment of time and work for little return (bel1997). This is because highly optimised code is usually difficult for human programmers to follow, and human programmer errors are the main source of software problems and development failures. Trying to create very complex optimised code often leads to code that does not work at all, and if it does work, is very difficult for others to follow, develop further, or simply maintain. In addition, the impact of trying to write highly optimised code on object-oriented languages is that good object-oriented design must be compromised. Object-oriented features of programming languages, especially inheritance and object encapsulation, are the main attraction to using object-oriented languages in the first place, but also the slowest aspects of these languages. A highly-optimised program produced using an object-oriented language would not in fact use object-oriented features. The benefits of the object-oriented approach would be lost.

In general, classes that perform well must start with a good design of an efficient and rational system. Poor design and planning is the main cause of fundamental performance problems, as demonstrated by the `Vector` example above. Object-oriented design should not be sacrificed. It is reasonable to expect the Java environment to run effectively with the use of objects, since these are its core components. The majority of the code will be kept in good object-oriented order. Some design decisions can address performance without conceding poor practice. For example, `StringBuffer` objects are mutable, while `String` objects are not. It is perfectly reasonable to use `StringBuffer` objects where string manipulation is required and cast into `String` objects thereafter. Performance will be improved without obfuscated or peculiar code. Techniques will be adopted to alter the programs only when necessary, when testing of the completed classes has identified performance problems that must be addressed. In this case the next step is to identify any performance bottlenecks and attempt to optimise these small areas of code. For example, approaches might include optimising loops, making methods and classes `final`, avoiding object creation and avoiding casting [wil2001].

The Java Virtual Machine can run a Java class and provide information on the time spent executing every Java method required, output to a plain text file. This profile can be analysed to show which methods and classes are utilising the bulk of processing time. This allows the small portions of code that contribute most to performance to be identified as candidates for optimisation.

There are two common techniques for improving performance that are not related simply to the code. One is to use a faster JVM. There are a number of JVMs for J2SE from a number of different vendors, all of which comply with Sun's JVM Specification, and they will differ in their performance. For example, since Java uses *compiled* byte code, not source code, a JVM should be able to compile the most commonly-executed byte-code to faster native machine code on the fly, a process called Just-In-Time (JIT) compilation [bik2000, kra1997]. Sun's HotSpot Virtual Machine [sun1999b] does this and claims significant performance improvements. However, a particular JVM cannot be forced upon end users of this system, and so relying on a particularly fast JVM is not a solution to poor performance of the translation classes.

The final technique is to use a different compiler. While all compilers produce valid Java byte-code, they may produce byte-code of differing efficiency. For example, the Sun Java compiler Javac can optionally perform some additional optimisations, like inlining `final` methods to produce faster classes at the cost of slightly larger class files. However, modern Java compilers have gone through many years of development and improvement and can be considered broadly similar. The program design is likely to be the main determining factor in performance.

Possible performance issues are a concern for every Java development. This project will measure the relative success or failure of the implementations with this in mind.

3.2 Addressing language universality

A key limitation of the current BrailleTrans implementation is the 256-character set limit on supported characters. This has repercussions for its universality and portability, and an alternative should be identified.

Computers operate only with numbers internally. Any representation of a human alphabet will involve a correspondence between characters, symbols with a fixed meaning, and numbers. All of the existing character encoding schemes rely on such a correspondence. Note that the relationship is between symbol *meaning* and number, not symbol *appearance*. For example, there are as many ways of displaying the letter 'A' as there are fonts and display devices, but they all mean "letter A" and will be encoded as such, for example, as number 65. A particular encoding scheme is referred to as a *character set*.

There are numerous character sets. First, 256-character sets must be explained.

3.2.1 Seven- and eight-bit character sets and ASCII

The way that characters are encoded in computer systems reflects the way that computers work and the history of their development. Early computers were severely short of memory, both short- and long-term store, which encouraged efficient space-saving character encoding. Computers were developed by English-speakers, who naturally encoded their own language. This lead to ASCII, the American Standard Code for Information Exchange. This is now an agreed international standard (ANSI X.3.4 and ISO646) (gsc1994) and has been the *de facto* lowest common denominator for computer representations of text for decades (fow1997). It is a seven-bit character set, giving 2^7 = 128 possible characters. These are assigned to the Western English character set, upper and lower case, punctuation, and a number of control characters intended for control of teletypes and file formatting. The majority of these control characters are now redundant, but the code for letters and punctuation is universal. ASCII has the benefit of being 7-bit, which means that a normal 8-bit byte can contain an ASCII character plus a parity bit for error correction.

Computers grew to use an 8-bit byte as the standard atomic unit of data, storing ASCII as 8-bit bytes. This provides an extra bit, which was utilised to provide *extended* character sets of 256 characters. The first 128 numbers are standard ASCII characters and the next 128 are system-dependent. Different operating systems and applications used these upper 128 numbers differently to represent different characters, which gave rise to a great number of character sets. Eventually a standard was agreed, and ASCII Number 5 became an American National Standards Institute (ANSI) standard in 1977 and was adopted with minor revisions by the International Standards Organisation (ISO) as ISO 646 (fow1997). This was expanded to create a number of ANSI character sets, all of 256 characters with the first 128 characters the same as in ASCII. For example, ISO 8879-1 or Latin-1, is the standard in HTML pages. The extra characters provided sufficient space to encode other European languages, for example Greek or French, but each language might now have a different character set, and the character codes were not reconcilable into a single set in future since their values conflicted. 256-character sets have been the norm until very recently, and even now are still the most common way to encode text on a computer.

The current translation implementation assumes the use of one of these 256 character sets on the machine on which it is running. It also assumes that the language table it is set to use has been constructed under the same character set. It will then perform translation efficiently and accurately, but it works not with the characters themselves, but with the integer values that are used to represent them. If, for any reason, the character encoding used by the composer of a language rules table differs from that of the input text passed to the program, translation will occur but the results will be unpredictable. This is a major barrier for portability between different operating systems and between different languages. The same character can mean many things:

Western name	Western set	Russian set	Thai set	Integer value
a acute	Á□	б	ม	225
c cedilla	Ç	з	ว	231
e circumflex	Ê	к	ส	234
o grave	Ò	т	า	242
n tilde	ñ	с	ั	241
u umlaut	ü	ь	๘	252
pound sign	£	Ј	ฃ	163

Working solely with 256-character sets does allow for some efficient and simple code in the current implementation. The character rules held in `ch_info` can be neatly defined in a simple array since their exact number is known in advance. Every character found in the input text must map to one of these rules since every character value 0 - 255 has an assigned mapping. More important is simple correlation between one 8-bit byte and one text character that follows from using a 256-character set. This greatly simplifies many of the computing problems posed by translation. Characters and their integer equivalents can be used interchangeably, as efficiency requires. Simple pointer arithmetic allows arrays of text to be navigated by simply incrementing or decrementing the pointer. The impact on parsing the translation rules tables from simple user-edited text files into compressed machine format, and importing the machine files into the translation engine, is considerable. No provision need be made for the possibility that a character might be represented by more than one byte. Characters may be read straight from disk as bytes. (The documentation of Mk asserts that escape-sequence characters may be used in the language rules tables, where `\xAA` would indicate the hexadecimal integer value of a character. This would still create only characters in the 256-character set, but allow them to be represented on systems where the correct character could not be displayed in the editor used for the tables, so the correlation still stands. In any case, this functionality is not implemented in Mk).

Despite these development reasons for continuing to support only 256-character sets, it would be desirable to develop an implementation that is not so limited. This could then be used accurately no matter what the local system uses for encoding. It could also support character sets that are not 256-character 8-bit sets, such as Korean. Attempting this in C would require the development of many string and character routines to cope with the different character encoding methods required. Java, however, uses an international non-256-character set internally and provides many classes for performing just these manipulations. First, this international character set used should be explained.

3.2.2 Unicode

Unicode intends to allow a one-to-one mapping between every human character and a unique integer value, rather than the overlap that exists at present for non-ASCII characters. Unicode therefore provides for entire non-English alphabets and languages. It is also an open standard, managed by the Unicode Organisation [uni2001a]. It has been widely adopted for character text encoding in the computer industry. Microsoft, Sun, the World-Wide-Web Foundation and others have agreed with the international standards bodies to use Unicode in future developments. Unicode is therefore an excellent candidate for character sets that are not contained within an extended-ASCII set.

Unicode is essentially a 16-bit character-encoding mechanism. Unicode characters from values 32 - 126 map to the ASCII character set, so the Western alphabet and punctuation is common to both systems. Other 16-bit Unicode characters map uniquely to a single character up to value D800h. A limited range of Unicode characters is used for expansion into a 32-bit character set that is still consistent with 16-bit Unicode. Unique mapping prevents the overlapping and ambiguity that accompanies the 256-character sets. However, the adoption of Unicode by the dominant US computing industry meant the use of twice as much storage space was required for every character. There will not be of much benefit for most American developers, since the US English alphabet is supported fully by the ANSI 256-character set. Therefore, flexibility on encoding Unicode was required to encourage takeup of the technology. A variety of Unicode Transformation Formats (UTFs) were therefore developed to encode Unicode character values (uni2001b).

UTF-8

Used on the Web and for Western English character set encoding. An eight-bit encoding mechanism, values 0-126 map to conventional ASCII. The most significant bit is therefore zero for all these values. This means that it is identical to ASCII for most programming uses. Values above 128 are held in numerous bytes, as many as required for the character value. Multiple byte characters are indicated by non-zero first bits in a mechanism that orders the bytes and allows a reading computer to determine which eight-bit byte or set of bytes maps to which character. This makes it space-efficient for Web and programming applications.

UTF-16 and 32

UTF-16 is used in Java, Windows NT4/2000 and XML, UTF-32 in Unix. Superficially each is simpler than UTF-8, being a 16- or 32-bit representation of each character. A clearer one-to-one UTF representation to character value mapping is obvious. However, a new complication arises, that of byte order within the 16- or 32-bit values. The two alternatives are known as "big-endian", or network order, and "little-endian", referring to the placing of the most significant byte at the beginning or end of the sequence of bytes. The program reading the string of bytes that makes up a piece of text must now know which of the two is in operation, or the results will be meaningless. This can be done reliably one of two ways. It can be assumed that the approach chosen is consistent throughout the environment, for example when a Java class passes a string in UTF-16 to another Java class, which will assume rightly the same UTF encoding. This is the best

> approach, but requires mechanisms to perform conversion from differently UTF-encoded text where this might be encountered. Good design is required to identify and cope with these situations. The other good alternative is the insertion of a value, or Byte-Order Mark (BOM) into the first position of the string that indicates whether the encoding is big- or little-endian. This is not desirable in storing and manipulating strings, where the BOM would have to be removed and added repeatedly and consistently. A third option, applying rules to try to determine the encoding of a piece of text, is not a good solution. It involves a larger overhead on processing every piece of text, and will not be completely reliable because the possibilities of encoding Unicode characters do not allow certain discrimination between the three forms of Unicode encoding.

A further complication of Unicode encoding is that even if text characters are consistent, the idiomatic control of formatting, most especially the control of *new-line* characters, is not standard [(dav2001)]. All might use standard Unicode characters, but Apple Macintosh machines use carriage return, UNIX machines linefeed and Windows machines both characters to indicate the end of a line. Even within operating systems different applications can use different codes. For instance, end-of-lines might be differentiated from paragraph breaks by the use of different control characters.

This is a limited problem for the translation system implementation. The translation engine and language, if designed as Java objects to be utilised as components as needed, will be passed Java string and character data, all encoded in Java UTF-16. Similarly, input text passed to the translation engine comes from another Java class, so will already also be in the Java internal encoding. The classes can rightly assume internal consistent UTF-16 encoding and ignore the issue.

However, input text has to be acquired from somewhere. Text is not produced in Java applications as a general rule, and it would be too great a limitation to require that all text to be translated be produced in a Java application. Text editors do exist for the majority of platforms that allow text to be imported in the native encoding mechanism and saved as Unicode, which could be read straight from disk and passed to the translation system. Again, this is a great limitation on use. It also does not solve the problem of text that must be encoded in a non-Unicode fashion, for instance HTML text, which is the ANSI-based Latin-1 encoding. The interface to the translation system must be able to read text in the native encoding format for the platform in which it finds itself. Java, in fact, provides classes for just this eventuality. Several of the `java.io` package classes read Java characters from stream of bytes, whether from an internal, disk or network resource. `java.io.BufferedReader`, for example, allows the JVM to read Java Unicode characters from 8-bit input streams by translating from the native encoding. The Java classes do not have to know the local encoding, they can simply ask the JVM and correctly translated Unicode characters will be returned. In addition, BufferedReader also recognises the local newline character, and translates this into Java format, so solving the newline problem.

Another more fundamental problem is with translation rule tables. If Unicode characters are to be supported, the language tables must be redesigned to handle Unicode text. The current Mk program to produce compressed machine files will not suffice, since it works byte-by-byte. In this more specialised case, it is reasonable to require that the files *are* encoded in a set manner, one of the Unicodes (big- or little-endian). This will become part of the language rules table specification. `java.io.InputStreamReader` can handle UTF-8, big-endian and little-endian Unicode, with or without an BOM. Any one of these encoding mechanisms would be valid.

Using Unicode characters is a valid extension of the translation system, and the Java library classes should make the problems this raises solvable.

3.3 Advancing the UMIST translation system: planned development

The technology to be used for the project has been chosen. It can now be stated what exactly will be developed to attempt to solve the problems identified with the current implementations. This is therefore description of the project deliverables or requirements.

3.3.1 Straight Java implementation with existing language files

This is an implementation of the translation mechanism devised at UMIST that will form a direct parallel of the existing C implementation, BrailleTrans. It might be regarded as a potential platform-independent replacement. As such it must utilise the existing translation tables, 256-character set-based, in their existing machine format. The choice of language file, provision of the text to be translated, initial machine state, and any formatting required will be left to the user, just as in BrailleTrans. Java 1.1 should be utilised, with the normal public class libraries used extensively, to maximise portability and maximise re-use.

Of course, supporting Java programs will be required apart from the translation engine. For testing and demonstration a Java interface to the translation system will be required that can feed input text to the translation system and display the results. Timing information for gathering performance data will also be required. A program to create authentic machine format translation tables from their text versions will be necessary for testing and debugging, to allow the contents of the translation tables to be altered and examined.

As a proposed replacement, the Java program should deliver a level of performance that when compared to the current C-code implementation suggests that it is an appropriate replacement, and that it can be used instead of the C implementation in a larger application. It will be surprising, by the nature of the languages used, to obtain equal performance. However, the performance of the C program is not the limiting factor in the Word-based UMIST system's performance, so the same level of performance as BrailleTrans is not necessary. In fact, in the Word-based system the Visual Basic routine that calls BrailleTrans for translation performs slowly and limits the speed of the whole system. In this example, the Java classes need only attain a level of performance that would not degrade the performance of the whole system. Equal performance is not necessarily required. Different coding approaches will be taken and the relative performance compared with each other and with that of the original C program. This will require extensive testing of speed and load on a number of machines differing in resources and operating systems.

It is to be expected that this work will form the basis of future development. For this reason the program should be fully documented, both within the code and with separate developer and user documentation.

Two approaches to the coding will be taken. The first is to convert the existing C program code into a Java class. This will not require major work. The mechanism for loading the translation rules tables will change, and the pointers used will become indices to arrays, but otherwise most of the code is directly usable because of the similar syntax and simple low-level algorithm used. Using procedural C code can be regarded as the least object-oriented and low-level implementation possible.

The second approach will be to use Java structures and objects to implement the UMIST algorithm. The same translation rules tables will be used. The two approaches will allow performance comparisons to be made and conclusions drawn about the best way to optimise the speed of the classes.

The Sun Java Native Interface (JNI) can be used to access native code executables, like BrailleTrans, from within the JVM [sun1997]. This provides an opportunity to further compare the performance of the Java classes with the C BrailleTrans implementation. BrailleTrans can be accessed from a Java class to perform translation and the output and time taken measured. This can then be compared to the performance of the Java translation classes. An amended BrailleTrans DLL can be constructed, which instead of performing translation simply returns the unprocessed result array. This will act as a placebo, so a comparison can be made between the performance of this DLL and the original BrailleTrans, both accessed through the JNI. This will ensure that the performance of BrailleTrans is being measured rather than the speed of the JNI mechanism.

3.3.2 Unicode-based translation system

The final implementation of the UMIST translation system should be an extension translation mechanism to utilise not the old 256-character-set translation system but Unicode-based language files, of no fixed character number, which can cope with languages outside the 256-character system. It will require changes to the translation engine and also to the translation rule table format, which will have to be in Unicode. A complementary program to convert Unicode text format language files into a new machine format, the task that Mk performs for the 256-character sets, will be required. This new machine format will simply be a Java object. A Unicode language rules table will be a Java object. This will provide a self-contained component that allows translation for a given language as needed.

Java Unicode language rules tables can then be `Serialized`, converted to data files, and stored on disk or on a network. Serialized language files can be compressed and stored as Java Archive (JAR) files by the Java `jar` tool. This reduces the size of the files, especially useful if they need to be transported over a network. It is the standard method for packaging classes and resources together in Java. They can then be distributed easily and quickly, but can be accessed by a JVM as easily as unpackaged files. The final distribution of the Java classes produced will be a JAR file.

Again, documentation and performance figures will be required for this development, so that it can be utilised fully in future work. Its performance will be compared with the implementation in Java and BrailleTrans accessed through the JNI.

3.3.3 Applications of Java implementation

The principle of developing a Java-based translation system as a Java class is that it can then form the basis of other more complete applications. This is comparable to the way that the BrailleTrans DLL is used within the Windows API to provide functionality within the Word applications. Java's object-orientation and universality should greatly facilitate such developments. To test this, exploratory development will be done in the following applications of the class. The limited results will used to demonstrate the potential utility of the UMIST system as a Java class. They can be regarded as "proofs of concept" rather than full implementations.

Enable a translation system to obtain language resources over a network using HTTP over TCP/IP.

Java is strongly network-centric and contains many native methods for accessing network-based resources. This should allow new or updated language files to be made available over the Internet and accessed as required. This would save users from having to obtain and install language files themselves.

Provide translation as a consumer-device application.

Java is heralded as a good consumer device development language, so the component might serve as the basis for a service on a personal consumer device. A version of J2ME will be selected and the Java 1.1 class adopted as necessary for this system.

Provide translation as a service in Sun's StarOffice word processor.

Sun provides the free office application StarOffice (sun2001f), with word processing, spreadsheet and other applications. It claims to provide a simple document model API, similar to that of JavaScript's Document Object Model, and allow Java applications to work with documents through this API. This would allow a similar implementation to that achieved by the existing system with Microsoft Word, translating StarOffice documents into Braille code.

Provide translation in a web browser.

A Java applet can be created that uses the translation class to provide translation functionality to a webpage through a web browser.

Provide an online translation service through a website.

The translation class can be accessed via a Java servlet, a class designed to be run by a Java server and provide server-side functionality to websites. A website providing translation should be possible.

It is possible that none of these attempts will be successful but details of the attempts themselves will illuminate the success of the Java-based strategy chosen.

4 Implementation of solutions

Several Java implementations of the UMIST translation system were produced to meet the requirements detailed in Chapter 3. They were each tested for performance, and attempts were made to use the implementations in prototype translation applications. The results of these tests are described in Chapter 5. However, the implementations themselves are of interest. A comparison of the different approaches, with detail of important elements of the implementations, will be of assistance in any future work on the UMIST translation system in Java. This Chapter therefore details and compares the different implementations.

4.1 Language interface

Java is an object-oriented language, so object-oriented programming techniques are generally very important in developing Java systems. However, this project aims to develop an implementation of the UMIST translation system that in itself forms only a component of a larger translation system. Object-oriented design in the context of this project, then, is limited to the ways in which the components developed would fit into a larger design. This simplifies the design considerations, but makes them no less important.

Object-oriented programming is a major and detailed subject. For the purposes of this project, a very limited explanation will suffice. An object-oriented system is composed of discrete atomic *objects*, which contain data and provide mechanisms to handle that data and services to other objects. They separate their public, documented, external interface to the world outside the object, from their private, internal implementation of the code within the object. Development then ideally consists simply of the combination of any number of pre-defined objects. These component objects are chosen by the developer according to what services are provided by the components, as defined in their public interfaces. The design of the Java implementations should therefore first define what their public interfaces will contain. Each of the implementations will be a different Java program, but each will have the same common public methods. A developer will then be able to use these implementations in a consistent object-oriented design.

In Java, the separation of the internal and external parts of an object is controlled by *visibility modifiers*. These are qualifiers on an object's variables and functions (*methods* in Java) that control whether the variable or method is internal, or *private*, invisible to other objects and developers, or *public*, part of the external interface for that component, to be used by other objects as part of a larger system. There are, in fact, four visibility modifiers in Java, because of the complexity of object-oriented features like inheritance. However, the main visibility modifiers used in this project are `public`, which makes the variable or method visible to any Java object on the system, and `private`, which makes the variable or method visible only within that object and inaccessible outside of it. The `private` variables and methods are the concern of the project only, and any future workers on the Java implementations' code. The `public` interface is the concern of any future developer seeking to use these implementations as they are to provide translation functions.

Defining this common interface across a number of different Java classes can be done with a Java `interface`. This is a prototype for a type of object that defines the common characteristics of all objects of that type. It defines a number of methods that must be implemented by every class that is declared to be of that type. A class of that type can then be regarded by a developer or another class as having all the methods and variables defined in the interface. It does not restrict what other `public` methods can be implemented, nor what `private` methods are used to provide the functionality required.

In technical terms, interfaces take the place of powerful multiple inheritance in C++, which was regarded as a source of errors and confusion and forbidden in Java. A single Java class cannot subclass more than one other class. A proposed new class might share the characteristics of two existing classes, but it cannot inherit the characteristics of both of them. Only one class can be subclassed. However, any number of interfaces may be implemented. Interfaces can be freely implemented, since they simply define what an implementing class must be able to do, not how it is done. Therefore, defining `Language` as an `interface` rather than a superclass to be `extended` (subclassed) allows for future classes to subclass a different class entirely and inherit its methods and variables. For example, if a Java applet is to be produced that performed translation, it must `extend` or subclass `Applet` or `JApplet`. It cannot then also `extend` a `Language` class. This would be multiple inheritance, which is not permitted in Java. The applet could not inherit methods or variables from a `Language` class. This would break the link between all the `Language` classes. However, as an `interface`, not a class, `Language` may be implemented by this proposed applet. Interfaces allow a minimum set of requirements to be defined. They do not otherwise restrict a class from subclassing other classes.

Knowing the capabilities of the translation system and its potential applications, an `Interface` for the Java translation implementations was defined. It can be represented in a simple Unified Modelling Language diagram, which allows object-oriented programming artefacts to be depicted diagrammatically (omg1999):

```
Language
-------------------------------------
-WILDCARD_FLAG
-SPACE_FLAG
-WILDCARD_NONE
-WILDCARD_ONE
-WILDCARD_SEVERAL
-LEFT_FOCUS_DELIMITER
-RIGHT_FOCUS_DELIMITER
-RULE_OUTPUT_DELIMITER
-RULE_CONTENT_DELIMITER
-TABLE_DELIMITER
-RULE_BUFFER
+FILENAME_EXTENSION
+DATAFILE_EXTENSION
-------------------------------------
  boolean setState(int state)
  int getPermittedStates()
  int getState()
  String translate(String toConvert)
  int[] translate(int[] toConvert)
```

This illustrates several important points about the implementation, so the actual code is presented here in full:

```
public interface Language
{
  static final int WILDCARD_FLAG = 64;
  static final int SPACE_FLAG = 8;

  static final int WILDCARD_NONE = 1;
  static final int WILDCARD_ONE = 2;
  static final int WILDCARD_SEVERAL = 3;

  static final char LEFT_FOCUS_DELIMITER = '[';
  static final char RIGHT_FOCUS_DELIMITER = ']';
  static final char RULE_OUTPUT_DELIMITER = '=';
  static final char RULE_CONTENT_DELIMITER = 0;
  static final char TABLE_DELIMITER = '#';

  static final int RULE_BUFFER = 128;

  static String FILENAME_EXTENSION = "";
  static String DATAFILE_EXTENSION = "";
  static final char FILE_EXTENSION_DELIMITER = '.';

  boolean setState(int state);
  int getPermittedStates();
  int getState();
  String translate(String toConvert);
  int[] translate(int[] toConvert);
}
```

A class declares itself to be of the type specified in the `interface` by stating in the class declaration that it `implements` the interface. For example, a new translation class `MyNewLanguage` that complies with this interface will be declared with:

```
class MyNewLanguage implements Language
```

The `Language` interface is `public`. Future alternative implementations or systems can freely implement it. This permits future classes that implement `Language` to be developed freely. This encourages future implementations to comply with this specification. If they do so, they will be consistent with the implementations produced by this project. Maintaining a consistent public interface for all UMIST translation Java classes will help future developers who wish to use the UMIST system, since all the classes will behave in the same way from an external perspective.

Apart from a number of methods, `Language` also defines a number of *constants*, variables that will be available to all implementations. These are defined with the `static` keyword, which means that they available through the class rather than an instance of it, and with `final`, so they cannot be altered in subsequent implementations of `Language`. Declaring the variables as `final` has two effects. The first, less important effect, is that this allows them to be *inlined* by the Java compiler. Their representation as variables in the code will be replaced by their literal value. This makes the compiled code faster to run.

The second effect is more important in object-oriented design. `Final` variables cannot be altered by any future class that implements `Language`. Like C `#define` constants they have a fixed value, defined when the variable is first declared. Unlike C constants, these variables will be available, still fixed in value, in every implementation of `Language`. The choice of what to define as a constant therefore helps to shape future implementations of `Language`. For example, the `int` wildcard flag is defined as 64 (01000000 in binary in an eight-bit flag byte), so every `Language` implementation and language table file has this available as a constant. Similarly, the whitespace bit is set to 8 (00001000), and the all the characters used as delimiters in the translation rule definitions are defined in `Language`. All derive from BrailleTrans and the current language rules tables. They will be used in all this project's implementations so there will be consistency across all the classes. Providing constants encourages future implementers to follow suit, which will encourage consistent language rules table formats. This aims to reduce a proliferation of different formats that would make different implementations of the UMIST translation system less universal and cross compatible. A universal set of standards for the language rules tables will make the system easier to support and promote.

Some constants determine internal hard-coded values that are the result of design decisions on resource allocation. In `Language`, `RULE_BUFFER` provides the size of array that 256-character-based implementations should use to read in one line of the translation rules table file. It therefore assumes that no single translation rule will be more than `RULE_BUFFER` characters in width - 128 in fact. This is a hard-coded constraint, a reasonable attempt to use the finite computer resources efficiently based on analysis of its role. For example, it is most unlikely that any 256-character set language requires a rule so long that 128 characters are not enough - none of the current rules for Standard British English is longer than 32 characters. Using a larger array "just in case" would therefore take up more memory than is necessary to cope for an extremely rare occurrence. Design decisions involved in choosing values for constants like these are often required in an application. The output text size ratio to input text size in BrailleTrans is an example already encountered. There exists a trade-off between efficient use of system resources, and performance, and no obviously correct value for the constant. Similarly, in BrailleTrans, the translation rules array, state table, wildcard array are among the data structures that use fixed-size arrays of a size judged to be appropriate for their contents but not too demanding on space. `RULE_BUFFER` in `Language` reflects similar decisions, and others will be highlighted later. It is the case that a Java implementation *may* use fewer constants than, say, a C program. For example, having to decide in advance on the size of data structures like arrays is a very common trade-off between performance and resources. Bigger arrays are less likely to become full and require time-consuming resizing. Smaller arrays take up less memory. Java, however, provides a number of data models, like `Vectors`, that allow constraints on size to be ignored - in theory. Although Java data structures may alleviate such hard-coded constraints, their relative inefficiency compared to simpler data structures like arrays can prohibit their use. As was seen with Vectors in 3.1.4 the judicious use of specific values is still required in Java programming.

The split of language rules tables into a human-editable text format and a compressed machine-readable format has been preserved in all of the implementations. This has not been retained simply for compatibility with the existing language rules table files. It is a good design feature that allows for easy editing and creation of language files by users with limited technical abilities and tools. The project introduces new types of language rules table files for the Unicode implementation. To allow these files to be differentiated easily, they have defined, different *filename extensions*. These use the '.' character to delineate the last letters of the filename. These letters indicate the type of data that the file contains, for example `txt` for text files or `htm` for HTML files. The `Language`interface therefore defines two `String` filename extensions for each class. These `static` variables should be replaced by the filename extension chosen for the human-editable language files for that class - `DATAFILE_EXTENSION` - and the machine-readable language files for that class - `FILENAME_EXTENSION`. They will then identify the correct human or machine format translation rules table files for that particular implementation:

Implementation	Type	Human format language file	Machine format language file
BrailleTrans	Legacy 256-character	`CON`	`DAT`
Language256	Legacy 256-character	`CON`	`DAT`
LanguageInteger	Legacy 256-character	`CON`	`DAT`
LanguageUnicode	Unicode	`UCN`	`ULF`

The filename extensions for the 256-character set implementations are those of the legacy 256-character files - they use the same language rules table files. New extensions have been used for Unicode language rules table files to differentiate the new and incompatible types. The concatenating ASCII '.' character is defined in `FILE_EXTENSION_DELIMITER`, to be used in conjunction with the file extension.

Finally, the interface defines the methods that all implementations should provide. This is the most important role of the `Language` interface. They are all *instance* methods, so they will be available to any Java instance of the class - any object made by instantiating a class that implements `Language`. `Language` itself cannot be instantiated because it is an interface.

`boolean setState(int state)`

> `setState` allows another class to choose the state of the finite state machine. The `boolean` return type is intended to indicate whether the method was successful. A failure will occur if an invalid state is requested, one outside the range 1 to (number of states).

`int getPermittedStates()`

This method returns the number of states permitted for the language. This should allow another class to avoid attempting to set invalid state values with `setState`.

`int getState()`

The last of the state control methods, `getState` returns the current state of the state machine. It is the complementary method to `setState`.

`String translate(String toConvert)` and

`int[] translate(int[] toConvert)`

These two methods provide the actual translation of text. The two perform the same function, but reflect the alternative ways to represent and process text.

The `translate` functions require more explanation. Unicode-based translators will work with `Strings` or `int[]` representations of text. Every Java `String` consists of 16-bit Unicode characters. Every one of these characters maps uniquely to a 32-bit `int` value. Therefore, a Unicode-based translation class can accept as input any `String` or `int[]` representation of text, so long as the characters represented are accommodated in the language rules table being used. Any character not defined in the language rules table will be mapped to whitespace. 256-character-set-based translation classes cannot translate characters with values in excess of 255. `Strings` passed to the class will be converted to `int[]`. Any value outside the 0 to 255 range will be mapped to whitespace.

In fact, all translation classes can be regarded as translating only limited ranges of the entire set of possible character values. Unicode-based classes translate an undefined range, while 256-character classes translate the range of values 0 to 255. The real difference between them is that a Unicode-based class can be defined in terms of the unique Unicode *characters* it translates. A 256-character-set-based class can only be defined in terms of the range of *integer values* it translates. The actual *characters* it translates depend not solely upon the language rules table used but the local encoding of characters to the 256-character set.

The interface provides no prototype for the constructor. In the implementations, the constructor takes the place of the `InitBrl` function in BrailleTrans. It creates a new `Language` object and loads a language rules table file from disk. Interfaces do not provide constructors, even as prototypes. In any case, there is no reason to restrict future instantiation of a class that implements `Language`. Many classes define `final private` constructors to prevent subclasses adding constructors later. These subclasses could then instantiate themselves, which may violate an object-oriented design predicated on the class and its subclasses *not* being instantiated. The `Language` interface, however, is intended to be flexibly subclassed and instantiated. It is intended as a versatile component of a larger language translation system.

Different implementations may require different constructors. For example, an applet might be created that implements `Language`. It would need to be instantiated with the correct applet constructors to function, not any defined `Language` constructor.

All the `Language` implementations, therefore, possess common public methods and variables. In addition, in every implementation the UMIST translation system is used to provide the methods of this public interface. However, every translation will differ in how the UMIST translation system is implemented, in representation of the language data, performance, and character sets handled. The three implementations are therefore of interest:

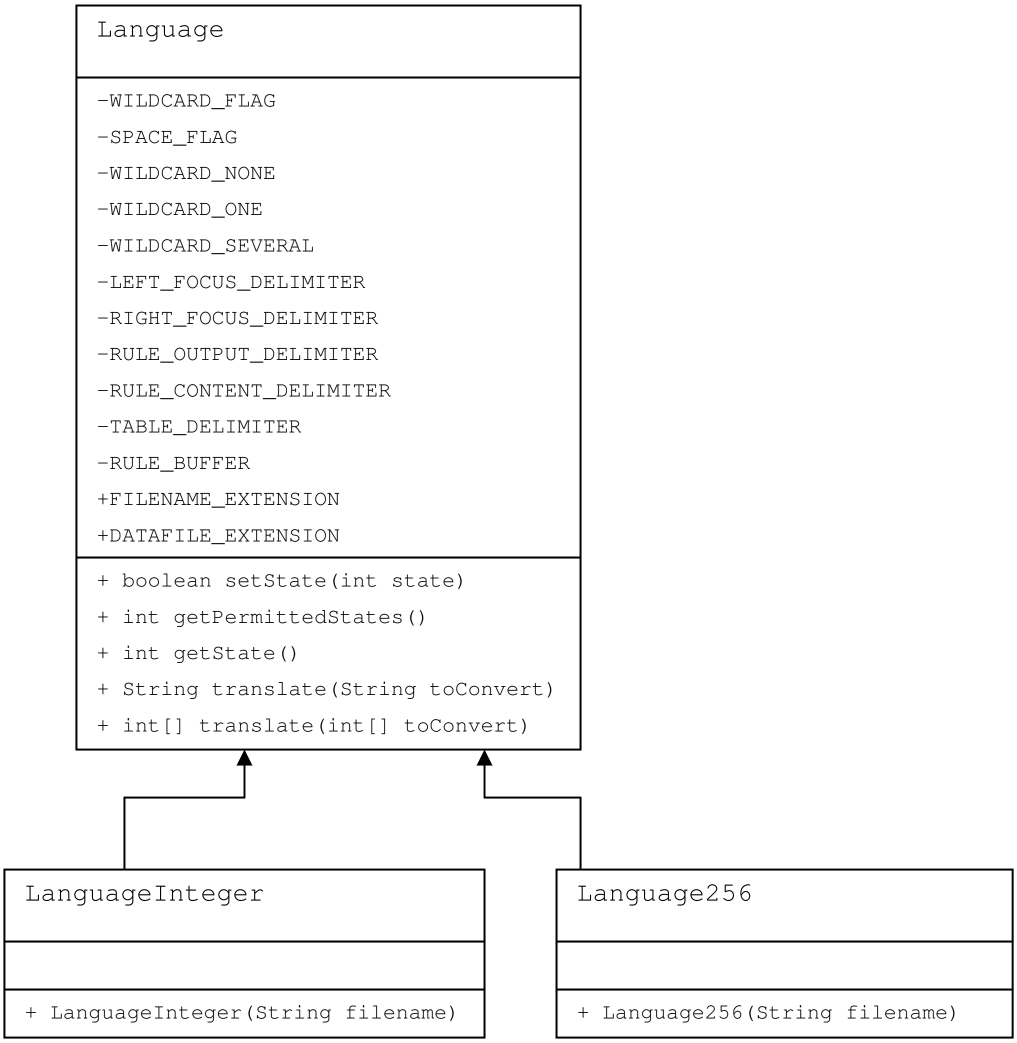
- LanguageInteger and Language256 use the existing 256-character set language rules tables and work on 256-character set text. They are detailed in Section 4.2.
- LanguageUnicode works with 16-bit Unicode characters in translation tables and input text. It is detailed in Section 4.3.

4.2 The two 256-character implementations, LanguageInteger and Language256

These two classes both implement `Language`. They deal with extended 256-character sets, and use the language rules table files in the same legacy format as used by BrailleTrans. This makes them backwards-compatible with the existing translation work.

- **LanguageInteger** is a direct porting of the current BrailleTrans C program. It uses the same representation of the language information internally, and almost exactly the same implementation of the UMIST translation algorithm.
- **Language256** is an implementation that seeks to use Java data structures instead of C-style arrays to represent the language information. Its implementation of the translation algorithm is based on these different data structures and has more in common with `LanguageUnicode` than `LanguageInteger`.

They can again be modelled using the UML layout:



As can be seen, both inherit the protected constants and public methods from Language. This ensures that they behave consistently, so each can be used where the other is used. The constructors, described below, add a public way to instantiate the classes.

Both have a number of common features that should be noted before each is examined individually. Both deal with 256-character sets. In BrailleTrans these are represented by the `unsigned char` type, which is a byte providing the same range of integer values as the characters with which it works. Java does provide a `byte` type, and there are reasons for using 8-bit `bytes` rather than 32-bit `ints`. Less memory would be required. The byte streams provided by the Java input/output disk access library classes might be handled more efficiently, since the underlying stream of information is a byte stream, and using `bytes` rather than `ints` would not need casting up to 64-bit `ints`. However, the Java `byte` is signed, which would cause some programming complications. A `byte` value outside of the range 0 - 127 (the ASCII range) would be negative in Java. It could not therefore be used for an array index, or cast to a `char` for inclusion in a `String`. However, these complications are not major and the internal processes of matching values and bit comparisons would work unchanged. The decision to use `ints` rather than `bytes` was based on three reasons. First, an `int` can always represent a Unicode or Java `char` but a `byte` cannot. If `bytes` were used for character data, then the `Language` interface could not then define two complementary and interchangeable methods, which is a less elegant design. Second, and a more practical reason, the use of `bytes` would require more resource-hungry casting and extra processing. `Bytes` would need to be repeatedly cast to `ints` so they could be used as array indices, and casting is inefficient in Java. The situations where casting was required would have be identified and correctly coded, introducing more complexity and a greater risk of error. More code would need to be developed. Thirdly, and most importantly, the porting of BrailleTrans to `LanguageInteger` would require significant changes to the BrailleTrans code. `LanguageInteger` proved very useful because another developer, independent of the project, coded it. It was therefore valuable as a standard, allowing the output of the other, original, Java classes to be compared to a completely different approach. This showed up several errors in the logic of the new Java classes that could otherwise have only been identified by lengthy testing. Translation results would have had to be manually calculated from close examination of the language rules tables and compared against the actual output produced by the classes. This detailed testing needed to be done eventually, of course, but differences in output between LanguageInteger and the other two classes highlighted several errors early. Early detection of errors in code reduces development time considerably. This benefit would have been jeopardised had significant changes been made to the internal workings of BrailleTrans to turn it into `LanguageInteger`. It could no longer have served as a control.

Loading the language rules table file from disk is therefore based on reading a sequence of `ints`. The classes of the `java.io` package are used. The constructors for both the `LanguageInteger` and `Language256` classes take as a parameter the full path and filename except the file extension - `String filename`. This path and filename has the appropriate filename extension appended to it, and is then used to create an input stream to read the data from disk:

```
BufferedInputStream inFile = new BufferedInputStream(new

  FileInputStream(filename));
```

Constructing the path and filename is the responsibility of the class creating the instance of the `Language` class. The path and filename will vary according to the local filesystem conventions.

For example, the String `filename` might be on a Microsoft Windows system `"c:\My Programs\Translator3\Languages\BritishEnglish"`, or on a UNIX system `"usr/alking/trans3/bin/data/britishenglish"`, or for a language file in the current directory, simply `"britishenglish"`. The syntax is immaterial to the `Language` class: it need only point to a valid language rules table file when passed as a parameter to the `java.io.FileInputStream` constructor.

A `FileInputStream` produces the stream of `ints` from a language rules table file that will be parsed by the class into the internal data structures making up the language. File input/output (I/O) relies upon the operation of the filesystem, outside the realm of the JVM. Accessing the filesystem is far slower than any internal operation. Optimising any I/O operations is therefore crucial to obtain good performance. A feature of disk I/O is that data is most efficiently obtained if large segments are read from disk infrequently, rather than small segments read frequently. It is very inefficient to read small pieces of data as and when they are required. Signficant performance improvements can be achieved by the use of *buffering*, where large segments are read from disk into memory and then accessed as needed. Only when the segment in memory has been exhausted does the disk need to be accessed again. The Java library classes provide buffered wrapper classes: in this case the simple byte stream `FileInputStream` is wrapped in a `BufferedInputStream` object. This provides the faster buffered performance.

Disk access is required only to load in the language rules table file when instantiating a `Language` object. No subsequent disk access is required for any amount of translation while that `Language` object is available. A developer should therefore seek to instantiate a `Language` object only once, and then complete all the translation necessary using the object. In general, a developer will try to minimise repeated object creation anyway, because it adversely affects performance, so this merely supports established good practice.

4.2.1 LanguageInteger

`LanguageInteger` is a port of the existing BrailleTrans C program into Java. Because of the common syntax and array structures, this is relatively simple for this program. The core translation function does not use common C libraries or platform-specific or hardware features. It is therefore unnecessary to find analogues of specific C entities in Java. Arrays are treated in a very similar way in both languages, and form the majority of the data structures in BrailleTrans.

After stripping out code specific to the implementation of BrailleTrans as a Microsoft DLL, three tasks remained. The first was to replace the C I/O functions with Java `BufferedInputStream` class methods. The second was to amend the C data types to reflect the new Java environment. unsigned chars became Java ints. The C variable modifier register, indicating a variable to be held in a fast processor register, could be removed because in Java such memory management is solely the preserve of the JVM. These were relatively trivial tasks. The third task was to replace the *pointers* used in BrailleTrans with Java operations. This was a more significant task. When C uses data structures like arrays it directly accesses the area in memory where the structure resides. Manipulation of the references or pointers to these memory areas allows C to perform pass-by-reference, rapid array access and other powerful operations. Java has no pointers, and uses objects and references for every data structure except the primitive types. The C code using pointers had to be translated into Java code that used object references. In fact, this proved relatively simple. The relationship between the C operations and Java operations is detailed in the following translation table:

C construct	Corresponding Java construct
Pointer prt to array myArray	Integer index index to array myArray object
Value referenced indirectly by *pointer	Java array member myArray[value]
Pointer value ptr	Array index index
Pointer address&ptr	Array index index

Of course, a direct conversion is not possible. Each translation must be examined to confirm that the operational logic is consistent between the related constructs and that the code will still fulfil its logical requirements. Examples of the translations performed in LanguageInteger demonstrate that this did not in fact prove too difficult:

BrailleTrans	LanguageInteger
Pointer looking to translation rules array table	Integer index looking to array table object
Entry in translation rules array *looking	table[looking]
Amending pointer value looking++	Amending array index looking++
Initialising pointer with address of item in memory looking = &table[ch_info[input_dat[up_to]].hash]	Initialising index with array index value looking = ch_info[input_dat[up_to]].hash

More amendments to the C code were required for error handling. BrailleTrans possessed a number of error codes, defined as integer constants. These were returned by InitBrl and Translate and indicated the success or failure of the functions. If an error had occurred the error code would help to identify it. This system of error codes has been maintained in

LanguageInteger, but the Java mechanism of *throwing exceptions* has replaced the simple return value. Java has a built-in error reporting mechanism. Errors generate an object of the `Exception` class. The Java compiler forces programmers to write code to handle or explicitly ignore exceptions at any points in the source code where they might be generated. For example, many methods in the `java.io` classes throw `IOException` errors or subclasses of them. Failing to find a named file will throw an `IOException` type, `FileNotFoundException`. (Some errors do not have to be handled because they are regarded as terminal, such as `VirtualMachineError`. These derive from the `java.lang.Error` class rather than `java.lang.Exception`. Others are thrown when an error occurs but do not have to be explicitly handled, like `ArrayIndexOutOfBoundsException`. Java error handling is a sophisticated system that can only be outlined here). There is no run-time overhead in defining these errors or the code to handle them. They are therefore an efficient way to write programs that handle potential problems elegantly. They should not be used in the place of normal statements used to prevent errors occurring (by checking array index values before the array is accessed, for example) because if errors do occur then significant run-time overhead is incurred.

The `Language` classes are all components, designed to form part of larger translation systems. They should not, therefore, pre-empt the implementer by handling fatal errors internally. Errors generated in the program are all thrown out of the program. The class that calls the `Language` class can then deal with any errors that arise as however the implementer chooses. Most of these errors thrown result from an error of disk I/O, and are standard Java `IOExceptions`. Some, however, relate to mis-formatting encountered when a legacy data file is being loaded from disk. This situation was documented in the original program by the error codes. For the Java `LanguageInteger` a new `Exception` was defined:

```
public class LanguageLegacyDatafileFormatException extends Exception
{
  public LanguageLegacyDatafileFormatException()
  {
    super();  // calls Exception() constructor
  }
  public LanguageLegacyDatafileFormatException(String description)
  {
    super(description);  // calls Exception(description) constructor
  }
}
```

This class extends or subclasses `Exception` and is thrown when mis-formatting is encountered while loading the legacy language rules table file. Its `String description` constructor is used to pass on a message indicating the problem area. Part of this message is the original BrailleCode error code. For example, when processing the version number of a language rules table file to check that it can be used, the exception is thrown if the version number fails to match and the language rules table cannot be used:

```
if (inFile.read() != 17)
    throw new LanguageLegacyDatafileFormatException("Language file"
      + format error, wrong version number. Code=" + VERSION_FAULT);
```

An implementer working with LanguageInteger can therefore write code to handle this exception, and can use the error code to help identify the problem.

In addition to the translation of the C code to the Java environment an important new feature had to be added. The C program, as described in Section 2, permits attempts to check input text outside of the boundaries of the array containing the input text. An array of size N might be queried for elements of index N or more or even indices less than 0. These attempts can occur during the comparisons of focus, left context and right context with the input text. In all of these cases, the input text is iterated through until a comparison with the translation rule item being compared is completed or failed. At no time is the index of the input text checked for validity. Java will not permit this behaviour - a run-time error will be generated if an attempt is made to access an element outside of an array. This is not satisfactory. The index to the input text must be checked before it is used to prevent this error. If the index is indeed found to be outside the array boundary, the comparison operation cannot proceed normally. It would be simplest to terminate the comparison operation and fail the attempt to match outside of the input text, but a better result is to regard all text outside of the input text array as whitespace. A segment of input text will likely have come from a larger document, so regarding it as being surrounded by whitespace is appropriate. The British English language rules table maps every non-text control character, including linefeed and carriage return, to whitespace. Whitespace is therefore the default for any undefined input text. With these examples in mind, LanguageInteger, and Language256 and LanguageUnicode later, were implemented to assume that all text "outside" of the input text consisted entirely of whitespace characters. This allows context whitespace wildcards to match correctly against the beginning and end of input text segments. For example, contraction rules that apply to the beginning of words will not match the first word of an input segment unless the text to the left of that first word is regarded as being whitespace. If it is, then the existence of a new word beginning at the first character of the input text segment will be indicated by the whitespace and the rules will match.

The last amendment made to the C BrailleTrans code was to implement the state control methods of `Language`. It was a minor change. The simple `getPermittedStates` and `getState` methods were straightforward and simply returned the value of the requested private instance variable. `setState` was also simple to implement. It sets the `defaultState` private instance variable to the requested state. When translation is subsequently initiated the state of the machine is set to `defaultState`, and in the event of a character in the input text failing to be matched with any rule the state again defaults to `defaultState`. A failure to set the state of the translation machine was coded to result in an error message being output to the `err` output stream and a `boolean false` value being returned. The state remains the same. What the calling class does with this result, and whether the `err` stream outputs to anywhere - typically it outputs to the console - is up to the implementer of the calling class.

The BrailleTrans C original represents a language internally with primitive data types, arrays, and one data type for the character data. When translated into Java for `LanguageInteger` this data type becomes another class, `ChInfo`. This class was defined as an *inner class*, which means that in design and coding it can be regarded as a private entity existing solely within the scope of `LanguageInteger`. Its methods and variables can be accessed only as private `LanguageInteger` methods and variables. Inner classes allow for simpler and more elegant source code, but are in fact resolved by the compiler into separate Java classes with accessor methods. They are an aid to good object-oriented design. The only other data structure used in `LanguageInteger` is another inner class called `Output`, used for the character rules and output text. Both `ChInfo` and `Output` are simple objects without methods, and act simply as containers for their data. This reflects the non-object-oriented, simple origins of `LanguageInteger` in the original C-coded BrailleTrans.

`LanguageInteger` therefore contains few objects, no casting, and no advanced Java library data structures. The two objects that are used have no methods, so costly run-time determination of dynamic method over-riding is not required. Like the original C program, the translation rules, wildcard rules, and state table are all stored in simple one-dimensional arrays. This imposes some hard-coded limits on the maximum size of these structures, and the same values have been used as in the original: 25,000 characters in the translation tables, 200 in the other arrays. (This is a limit on the number of items that can be stored in each array, not the number of bytes each array takes up. The number of bytes an item occupies in memory in Java depends on the JVM). `LanguageInteger`, therefore, utilises none of the features of Java that provide its rich programming environment while at the same time degrading its performance. It is likely therefore to be the fastest Java implementation of the UMIST translation system developed by this project.

4.2.2 Language256

This second implementation of the `Language` interface was entirely developed in Java. Like `LanguageInteger`, all characters are stored as `ints`, and again, the assumption is that character values will be in the range 0 to 255. This permits the use of the legacy language rules tables. The `Language256` constructor is responsible for loading and parsing the legacy language rules table. It is passed the path and filename for the language file and appends the correct filename extension to this path before loading the file from disk. All this is identical to `LanguageInteger`. The first differences between the two 256-character set classes relate to the internal representation of language data.

The state table is held as a two-dimensional array of `booleans`, of dimensions (number of states times number of input classes). The other data structures (wildcards, character rules and translation rules) are all represented as Java objects and implemented as inner classes. This allows more high-level logic to be used in the program implementation than would have been possible had `LanguageInteger`'s restricted and low-level internal language representation been repeated. The data structures are not, however, fully-defined objects in the sense that operations on their contents are fully part of their object definition. Rather, each serves solely as a container to hold the data in a logical and structured manner. Character information is held by a `CharacterRule256` object. Each instance of this class holds the mapped character, the upper-case version, and the character's flags. The upper-case version is not used in `Language256` (nor indeed in BrailleTrans or `LanguageInteger`, even though it is loaded into memory in both). However, the character information requires to be read from disk if only to be ignored, so for simplicity and potential future use it is read and stored. Wildcard information is held in similarly simple `Wildcard256` class. This contains the wildcard character, its flags, and the number of input class characters it matches, as defined by the `MATCH_ONE/MATCH_NONE/MATCH_SEVERAL` constants from `Language`. Finally, each translation rule is held as a `TranslationRule256` object. Rather than storing the rule information (left context, focus and so on) in an single array range, as the BrailleTrans and `LanguageInteger` implementations do, the `TranslationRule256` class defines a number of atomic values (for input class and new state) and arrays (for left and right context, focus, and output). All are composed of `ints`. This allows easy access to the constituent parts of a rule at the cost of some initial parsing. Since a language file need only be loaded and parsed once, when a Language256 class is instantiated, and can then provide unlimited translation through that instance, this should not be a significant cost.

These data constructs might have been made fully-fledged objects with methods for manipulating their contents. For example, the TranslationRule256 class might have had a boolean compareLeftContext(int[] input, int inputIndex) method, which would compare the rule's left context with the input array passed to it and return a result indicating whether or not the two matched. This might have produced more encapsulated, discrete objects and therefore a better object-oriented design. However, this approach was rejected during implementation when it became apparent that the methods required depended on access to all of the objects in combination. The compareLeftContext method given above would need to be passed the language's Wildcard256 data construct. This interlinking made object encapsulation too difficult.

In fact, two other design were considered for Language256, as the first Java implementation of the UMIST translation system. An initial plan split the language rules table information from the finite state machine, in two classes. This would be roughly equivalent to putting the translate methods of the final implementation into another class, TranslationEngine. This was rejected because the two classes were intimately interlinked to the degree that their separation was quite unnecessary. Neither was a feasible individual component or independent object by itself. The second plan resembled the current implementation, but stored the translation rules in a separate TranslationRuleTable class. Every Language256 would have a single TranslationRuleTable, which would store the translation rules and provide all the comparison methods using the rules to the Language256 methods. However, because of the interlinking described above, refining this design resulted in all of the language data structures being required to reside in the TranslationRuleTable rather than in the Language256 that contained it. This made the separate class superfluous. The current design has no such obvious weaknesses. Because it implements the Language interface it should be able to form a rational, discrete component of a future translation system.

The individual objects used for each wildcard, character rule and translation rule are stored in arrays (CharacterRule256[] for example). These are compact and fast data structures. However, the size of an array (the number of items that can be held simultaneously in the array) must be defined when the array is first created. The number of wildcards is defined in the language table and the number of characters is fixed at 256 for this implementation, so both of these arrays can be created with the correct size. The number of translation rules, however, is not so defined. In BrailleTrans and LanguageInteger, the translation rules are stored in a single array of `ints`. They define a maximum size for this array, of 25,000 characters. (The British English text-to-Braille language has 1351 translation rules, which allow for 18.5 characters per rule. The vast majority of rules in the language are shorter than this. The BrailleTrans value for the array size is another performance-affecting hard coded constant. It has been used in `LanguageInteger` for the same purpose). This array is populated by simply reading in translation rule data from disk until the language rules table file is exhausted. This allows a human language rules table composer to define the translation rules without needing to record how many rules there are.

`Language256` now follows this approach. However, an initial solution to this problem was developed which involved reading the translation rule information twice, once to count the rules and once to read them into the new array defined with the rule count. The language file was read with the Java library class, `BufferedInputStream`. This has a method `mark(int readlimit)` that allows the current position in the input file to be remembered and returned to later by use of the `reset()` method. The file was marked at the beginning of the translation rules, and the rules counted one by one from the input file until the end of the file was reached. The `BufferedInputStream` was then reset to the mark, and the `TranslationRule256` array created with the rule count. This worked, but suffered two problems. First, the `mark` method requires a parameter, the `readlimit`, which cannot be exceeded by the count of bytes read from the input stream before `reset()` is called. This means that the size of the remainder of the language file had to be determined. The `FileInputStream` method `int available()` returns the number of bytes left to read in the file. Since the `BufferedInputStream` being used was wrapping a `FileInputStream` this method was available. However, the Java API documentation warns that its result may be inaccurate, since it depends on accurate reporting by the operating system outside the JVM. The obtained value was therefore increased by 1000 just in case of under-reporting by the operating system. The second problem with this count-then-load mechanism was that it required more disk operations, which tend to be slow, and more processing in counting through the rules. The rules had to be parsed in a limited fashion to be counted, since they are of variable length.

For these reasons this mechanism was replaced by a simpler solution that assumes a maximum number of translation rules and declares the `TranslationRule256` array with this value. It then proceeds through the translation rules only once, parsing each rule in turn. This will result in an array larger than it needs to be to contain all the translation rules, wasting space, and also sets a hard coded limit on the number of translation rules for any language rules table. However, a Java array occupies in memory only the space needed to reference as many items as the array contains, not the memory needed to store every item. This latter memory allocation only occurs when the item is instantiated, in this case when a rule is obtained from the file. The overhead of spare array space is therefore minimal. The array size chosen, 2000, reflects a balance between wasting space and accommodating languages with large numbers of translation rules.

The different data structures require different handling than in TranslateInt/BrailleTrans, but aside from this the underlying logic and implementation is very similar. The Java data structures in Language256 made the programming task of implementing the UMIST system much simpler. Items like input classes or output arrays could be referred to directly in the code (`translationRule[transRuleIndex].inputClass` and `.output` respectively). If required, complex statements could be simplified by replacing a lengthy expression by a method call to a simple method that performs the expression. For example, the code

```
while ((inputIndex >= 0) &&
   ( flagsEqual(wildcards[thisWildcard].flags, input[inputIndex])) )
     inputIndex--;
```

comes from the `compareLeftContext` method. The code iterates leftwards through the input text until the current wildcard fails to match or the beginning of the input text is reached. It uses a method call to `flagsEqual` in the argument of the `while` statement. This is quite clear in the code. The code without the method call is more complex and difficult to comprehend and check for error:

```
while ((inputIndex >= 0) &&  ((wildcards[thisWildcard].flags &
  characterRule[input[inputIndex]].flags) != 0)
    inputIndex--;
```

Implementing the checking of left and right contexts, with variable wildcards and boundary checking, was the most challenging coding problem across all of the implementations, and gave rise to the greatest number of logic errors. Simplifications like the above were employed to simplify the code and make it easier to prove methods and debug. `LanguageInteger` uses a complex low-level mechanism, and while it was studied in detail and understood for this project it is unlikely that such a complex solution would have been possible had it been attempted.

Some observations about the differences between the `LanguageInteger` and `Language256` algorithm implementations are of interest. The `LanguageInteger` algorithm implementation, of course, is also almost entirely the BrailleTrans algorithm implementation. The `CharacterRule256` array and `ChInfo` array operate identically. The array is indexed by the character for which information is required. The object returned contains the mapping character, index of the first translation rule that might match it, and flag information. Mapping or normalising the input text is identical. Most of the comparison methods were very similar. `int` arrays are iterated through until a discrepancy is found or the item to be matched is exhausted. The difference lies in how exhaustion is determined. `LanguageInteger` items are exhausted when a delimiting character is encountered. From `wild_match`, comparing the right context:

```
while (table[looking] >= ' ')
{
  if (table[looking] == RULE_OUTPUT_DELIMITER && step == 1)
    break;
  /* CHECK FOR MATCH. RETURN FALSE IF FAILS */
  looking++;
}
```

In contrast, Language256 items are exhausted when the index to the array exceeds the array length. From `compareRightContext`:

```
int[] rightContext = translationRule[transRuleIndex].rightContext;
int contextIndex = 0;
while (contextIndex < rightContext.length)
{
  /* CHECK FOR MATCH, RETURN FALSE IF FAILS */
  contextIndex++;
}  // end of while contextIndex < input.length
return true;
```

The nested `if` statements in the `translate` methods to check focus, state, left and right context are also identical. One larger difference in the algorithm is the way that the translation rules are managed. In LanguageInteger, the `ChInfo` structure hashes directly into the large `int` array and navigates from there character by character. With each translation rule in turn, the focus is first examined to see if it still matches the current character in the input text. When it fails to do so there are no more rules for that focus in that language and the character should be skipped over - no rules have been found to match it. Language256, in contrast, utilises its translation rule objects by means of an `int transRuleIndex` that indexes the array of translation rules. This is set with the `CharacterRule256` array, which provides the index of the first translation rule with matching focus. When this is set, the index of the beginning of the next focus category is also obtained, and this value indicates when the focus category is exhausted. Until that point, simply incrementing `transRuleIndex` allows the translation rule array to be iterated through while searching for a matching rule. (Essentially the translation rules resemble an implementation of the `java.util.Enumeration` interface, which defines two methods, `hasMoreElements` and `nextElement`).

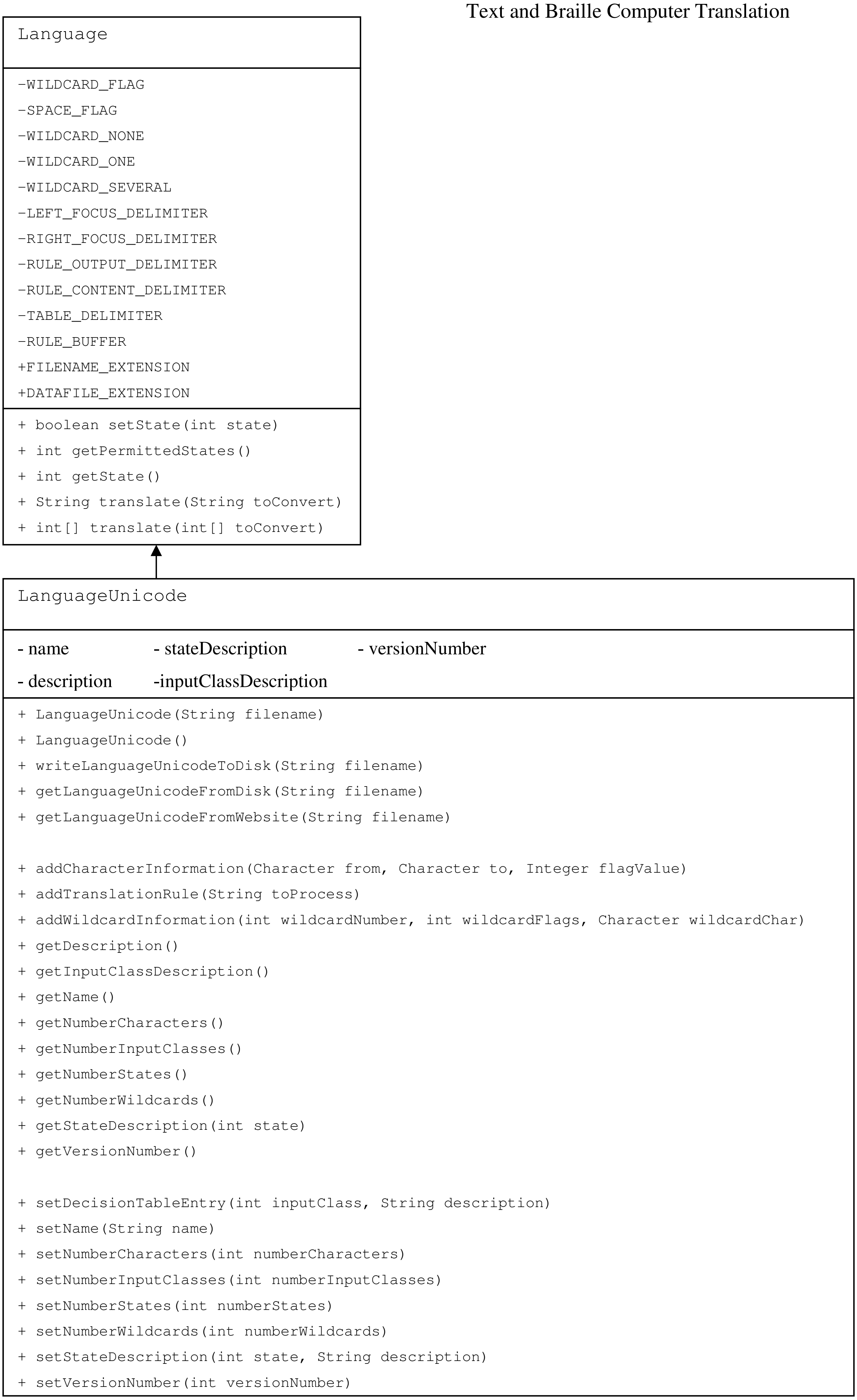
One difference of note has been mentioned already. Language256 allows for output arrays that exceed double the size of input arrays. This is achieved by pausing the addition of new output to the output array when the capacity of the output array is exceeded. A new larger array is created, based on the amount of input text left to translate and the size of the existing output array. The old output array and the remains of the output data that would not fit into it are copied into the new array. If the new output array is still not large enough, the mechanism is invoked again in a recursive operation until the output array is of sufficient size. This will not be a common occurrence - the MAX_COMPRESS value of 2 will suffice in almost all cases. The problem might occur with very short input sequences, as might be produced from the iteration of a translation system through a text document where a short title is encountered and sent for translation. This minimises the memory and performance risks of using a recursive operation - it will likely only be performed on smaller arrays. Longer stretches are less likely to be further from the mean expansion proportion. The code for the method is simple but illustrative of this recursive approach.

```
private int[] resizeConverted
  (int outputIndex, int[] outputContent, int[] converted,
   int remainderToTranslate)
  // outputContent[] - the output to be added that will exceed the
  //                   current output array size.
  // outputIndex - where in the outputContent has already been copied
  //               to the output array
  // converted[] - the old output array to be expanded
  // remainderToTranslate - how many characters remain to be processed
  //                        in the input text, including this rule's.
{
  // create a new output array, equal in size to the old
  // plus the size of the remaining text - including this
  // offending portion - times the MAX_COMPRESS constant
  int[] resizedConverted = new int[(int) (converted.length +
      remainderToTranslate * MAX_COMPRESS)];
  // check the new array is now big enough: if not, call this method
  // recursively with the new array to create a larger one still.
  if (resizedConverted.length <
      (converted.length + outputContent.length - outputIndex))
  {
    resizedConverted = resizeConverted(outputIndex, outputContent,
      resizedConverted, remainderToTranslate * 2);
  }
  // copy the old output array to the new output array
  System.arraycopy(converted, 0, resizedConverted, 0,
    converted.length);
  // copy the output still to be added to the new array.
  System.arraycopy(outputContent, outputIndex, resizedConverted,
    converted.length, (outputContent.length - outputIndex));
  // return the new expanded array with the added output.
  return resizedConverted;
}
```

One neater approach to the whole variable-size output array problem is possible. The `ByteArrayOutputStream` class in `java.io` allows `ints` to be written to an internal, growable `byte` array. No initial capacity needs to be defined, and there is no (defined) limit on adding to it. This was not used for two reasons. First, the resulting array is available back only as a `byte[]`, which would require a re-casting of the entire array and processing the signed byte values (-128 to 127) to positive unsigned `int` values (0 to 255). This would produce the required `int` array for output from `translate`. Another alternative would be to *pipe* the `ByteArrayOutputStream` into another Java input stream, which would allow `ints` to be read directly from it. However, the second problem still remains, which is that convenient Java library classes that provide this kind of functionality must be more complex and therefore may perform and scale less well than simpler solutions that involve more work on the part of the developer. This implementation is intended to be fast and simple. It already has the mechanisms to cope with typical and atypical situations, and so this alternative was objected.

4.3 LanguageUnicode

This last implementation of Language extends the translation system from 256-character sets to the Unicode character set. Again, it implements the Language interface with more additions than LanguageInteger or Language256. These reflect the task of building the LanguageUnicode object. The empty (no parameters) constructor and the many add, get and set methods allow a LanguageUnicode object to be constructed – by the MakeLanguageUnicode class described in 4.4 for example – and queried to find out its attributes and how it can be used. The object can be saved to disk using the writeLanguageUnicodeToDisk method. Finally, the object can be obtained over the Internet as well as from a local disk by the getLanguageUnicodeFromWebsite method. These processes are described below.



```
Language

-WILDCARD_FLAG
-SPACE_FLAG
-WILDCARD_NONE
-WILDCARD_ONE
-WILDCARD_SEVERAL
-LEFT_FOCUS_DELIMITER
-RIGHT_FOCUS_DELIMITER
-RULE_OUTPUT_DELIMITER
-RULE_CONTENT_DELIMITER
-TABLE_DELIMITER
-RULE_BUFFER
+FILENAME_EXTENSION
+DATAFILE_EXTENSION

+ boolean setState(int state)
+ int getPermittedStates()
+ int getState()
+ String translate(String toConvert)
+ int[] translate(int[] toConvert)
```

```
LanguageUnicode

- name              - stateDescription         - versionNumber
- description       -inputClassDescription

+ LanguageUnicode(String filename)
+ LanguageUnicode()
+ writeLanguageUnicodeToDisk(String filename)
+ getLanguageUnicodeFromDisk(String filename)
+ getLanguageUnicodeFromWebsite(String filename)

+ addCharacterInformation(Character from, Character to, Integer flagValue)
+ addTranslationRule(String toProcess)
+ addWildcardInformation(int wildcardNumber, int wildcardFlags, Character wildcardChar)
+ getDescription()
+ getInputClassDescription()
+ getName()
+ getNumberCharacters()
+ getNumberInputClasses()
+ getNumberStates()
+ getNumberWildcards()
+ getStateDescription(int state)
+ getVersionNumber()

+ setDecisionTableEntry(int inputClass, String description)
+ setName(String name)
+ setNumberCharacters(int numberCharacters)
+ setNumberInputClasses(int numberInputClasses)
+ setNumberStates(int numberStates)
+ setNumberWildcards(int numberWildcards)
+ setStateDescription(int state, String description)
+ setVersionNumber(int versionNumber)
```

First, the use of Unicode. The `translate` methods of Language support Unicode. Since all Unicode characters (16-bit) can map to a 32-bit Java `int` the two methods can be used interchangeably, but the int[] method in fact converts the array into a String (casting each array member to a `char`) and the String output back to an int[], so the String method is to be preferred for performance.

The class implements the UMIST translation algorithm internally, of course. The data structures chosen for internal representation of the language data differ markedly from LanguageInteger and to a lesser degree from Language256. The methods that make use of these data structures to perform translation - `compareLeftContext` for example - do not differ markedly from `Language256` in logic. Again, the most significant differences arise from the structuring of the data and how the methods access it.

Like Language256, LanguageUnicode uses objects for translation rules and wildcards defined as inner classes. Again, these serve to hold data rather than provide fully encapsulated objects. Unlike Language256, LanguageUnicode also makes use of the `java.util.Hashtable` class, a Java 1.1 library construct. Key and value pairs can be added to and removed from the Hashtable. The key can then be used to obtain the value back from the Hashtable. These Hashtables are used in several ways in LanguageUnicode. First, one Hashtable is used to map individual characters in the input string to their converted values, and another Hashtable is used to to map individual characters to their unique flags. This removes the necessity for the existence of a character rule object, which in Language256 held only these two values. (Language256, which does use a `CharacterRule` object, stores the uppercase value for a character, as provided in legacy language rules table files. This is not the case in LanguageUnicode since there was no reason to take up space and time with this unused information. If more than two values were required for any one character, a character rule object might have been employed. As it is, the two Hashtables preclude the need for one). More importantly, it allows a variable number of non-consecutive characters to be used in the language. The number value of a Unicode character might be anything from 0 to 2^{16}-1. The characters may not form a contiguous block, certainly not from value 0 upwards. Because a Hashtable can have as few or as many characters as are required this allows LanguageUnicode to be entirely flexible with the number of characters a single language can support. Also, because Unicode should allow the coding of any character, LanguageUnicode should be entirely flexible with what characters can be supported. Another Hashtable is used to map wildcard characters with WildcardUnicode objects holding wildcard data. In all these cases the Hashtable is used to obtain simple data from a character.

The second Hashtable use is to allow an input character to retrieve the first translation rule that might match it - the function provided by `ChInfo` and `CharacterRule256[]` in the other implementations. LanguageUnicode uses a TranslationRuleUnicode class to hold a single translation rule's data, but does not arrange them in an array. Rather, the translation rules are formed into a number of linked lists. Each linked list shares the feature that the first character of the focus of each constituent rule is the same, that is, they are all in the same focus category. The first rule of the linked list contains the first-ordered rule in that category, and each successive rule is in the order of the original data table. A Hashtable is used to access these linked lists. Using the latest character from the input text as the key, the first rule is returned as the value. If it is found not to match, the next rule may be obtained from the first rule and so on until either a match is found or the focus category linked list is exhausted. Note that Hashtable works only with key/value pairs of Java Objects. This means that the key must be an instantiated object, so a wrapper class is used on the primitive (`Character` for the `char` primitive) to produce the key. In addition the value returned must be explicitly cast as a TranslationRuleUnicode object. Exhaustion is indicated by a simple boolean value, part of each rule, which shows whether the rule is the last:

```
currentRule = (TranslationRuleUnicode)
    tRuleGetter.get(new Character(first));
  // get the first rule of the focus category to examine.
boolean matchFound = false;
  // shows whether a match has been found yet

...

while (!matchFound &&  (!currentRule.lastInCategory))
  // repeat until a match is found OR no more rules in this focus
  // category exist, indicated by lastInCategory boolean attribute.
{
  currentRule = currentRule.nextRule;
    // get the next rule in the focus category

  /* TEST TO SEE IF RULE MATCHES */
}
```

These Hashtables are very convenient and elegant solutions, but there are performance drawbacks. In Language256 finding the Wildcard256 object that matches a given wildcard character is obtained by iterating through the wildcard array until the wildcard character that matches the character found in the context is found:

```
for
  (int thisWildcard = 0; thisWildcard < numberWildcards;
    thisWildcard++)
    // iterate through the wildcard array
  {
    if (wildcards[thisWildcard].character == leftContextChar)
      // match found.  wildcards[thisWildcard] is matching wildcard
    {
      /* PROCESS WILDCARD wildcards[thisWildcard] AGAINST INPUT TO SEE
         IF IT MATCHES */
    }
  }
```

A similar mechanism operates in LanguageInteger. This is simple and fast, at least for small arrays (larger arrays would need a better search algorithm). In fact, it is likely to be similar to how Java implements the Hashtable class internally. However, a Hashtable will perform more slowly than an array. It allows for more high-level functions, like non-unique associations, and variable sizes. It requires object instantiation and casting to obtain the values. It is therefore more complex. The larger the Hashtable, the greater the performance difference will become. It is unlikely that a Braille language will need to create entries for several thousand characters, but not impossible. There will therefore be some performance hit in using Hashtables. (A Hashtable would not be appropriate to replace the entire focus-matching process. It might appear that the translation rules could be replaced by a large Hashtable of focus string keys and translation rule values, so the matching translation rule for a given focus could be obtained immediately without needing to iterate through a list or array. However, a single Hashtable key must return only one value, and some translation rules have identical foci. It would be possible to return a small array of matching-focus rules, but this would not be easier to code and considerably slower in execution). It would be possible to implement an alternative to some of the Hashtables where only unique keys are used, like finding character flags or translation rule foci groups [shi2001], but the library Hashtables will be used for simplicity and rigour. Only if they prove to be a source of considerable performance problems will a replacement be coded.

In addition to the constant performance degradation of using Hashtables, there is a particular problem with their flexible size. When first declared, they must be provided with an `initialCapacity` value. They can be provided with an optional `loadFactor`, or if none is given, a default load factor of 0.75 is assumed. When the growing Hashtable reaches a size of (initialCapacity times loadFactor), an internal rehash operation is initiated. This expands the internal capacity of the Hashtable so that Hashtables of any size can be produced. The rehash operation is time-consuming, and should be avoided if possible, but additions to the Hashtables are not made except during language creation. There is no addition during translation. The only complication, then, is what initial setting of initialCapacity and loadFactor produces the best performance when no rehashing will occur. The Java API documentation suggests that the 0.75 default is a good level, and that higher values will increase the time taken to perform hash functions [sun2001b]. This is therefore used in LanguageUnicode. If subsequent testing indicates that the Hashtables are key performance bottlenecks this value may require further investigation.

In line with all of the other implementations of the UMIST system, LanguageUnicode copies an offending input text character which it has failed to match to a translation rule straight to the output text rather than outputting a whitespace character. This is perhaps the least defensible time this option has been chosen. In the 256-character based implementations every character should be in the range of possible characters for the language data. If it is not, then a Unicode character of numeric value in excess of 255 has been passed to the class. In this case the initial input text character mapping step will catch these characters and map them to whitespace. If every character in the input text is within the range of characters for the language, then if no translation rule is found it is for the reason that no matching translation rule exists. The character is not necessarily invalid. The most appropriate thing to do then is to copy the input character straight to the output character. This is different from when a completely alien character is found on the input text, when the substitution of whitespace is more rational.

One advantage of working with `String` objects in LanguageUnicode is that the problems of maximum array size do not exist. `Strings` can start at zero length and expand as far as necessary. In fact, a `String` object is not used in the class for the output text. This is because `String` objects are immutable in Java, so `String` addition in fact involves casting and object creation and is very resource-hungry. The `java.lang.StringBuffer` class should be used instead, declared with an initial size the same as the input text String. This prevents inefficient resizing as the `StringBuffer` output grows. New text may then be added with the `append` method without the overheads of `String` addition. In fact, `Strings` use `StringBuffers` internally for addition. This is one of the most common performance tips available for Java, but this is its only potential application in the program [wil2001].

LanguageUnicode keeps the separation of language translation into the engine and the language table file, but diverges from the other implementations in the nature of the language rules table file. The other implementations use the 256-character set based language files, which consist of compressed machine-readable versions of the human-editable language rules table files.

LanguageUnicode reads a fully defined LanguageUnicode object straight from disk. This is the equivalent of instantiating a Language256 class with a given language rules table and then storing a snapshot of that object to disk. It can then be recalled from disk and used directly for translation - the language rules table data is already part of the object, in its TranslationRule256 array, `boolean stateTable` and so on. The same process is applied to a LanguageUnicode object. This means there is no parsing process when a language rules table is loaded from disk. Instead, the constructor receives the path and filename for a LanguageUnicode object stored to disk, and uses the `java.io.ObjectInputStream` class to load it from the `FileInputStream`. Buffering is not required since an object stored to disk contains information on how it should be read back from disk efficiently.

```
public LanguageUnicode(String filename) throws IOException,
ClassNotFoundException, FileNotFoundException
{
  ObjectInputStream inObject = null;
  LanguageUnicode fromDisk = null;
  String objectFilename = filename + FILE_EXTENSION_DELIMITER +
    FILENAME_EXTENSION;
  try
  {
    inObject = new ObjectInputStream(new
      FileInputStream(objectFilename));
    fromDisk = (LanguageUnicode) inObject.readObject();
      // notice explicit cast required from Object to LanguageUnicode
    inObject.close();
  }
  /* CATCH I/O AND OBJECT EXCEPTIONS */

  this.version = fromDisk.version;
  this.numberInputClasses = fromDisk.numberInputClasses;
  this.characterMapper = fromDisk.characterMapper;
  /* ASSIGN THE REMAINDER OF THE ATTRIBUTES OF THE OBJECT READ FROM
     DISK */
  return;
} // end of LanguageUnicode(String filename) constructor
```

This is considerably simpler than the parsing required of the legacy language rules table files. As the constructor, the `this` keyword refers to the object being created, and it is used to assign the data structures of the object read from disk to the object being created, and implicitly returned from the constructor. This is a little cumbersome, since the list must be updated with every change to the attributes of the object. As an alternative a new method allows an object to be read from disk and returned. This could be used in place of the constructor:

```
public LanguageUnicode getLanguageUnicodeFromDisk(String filename)
  throws IOException, FileNotFoundException, StreamCorruptedException,
    ClassNotFoundException
{
  ObjectInputStream inFile = null;
  filename = filename + FILENAME_EXTENSION;
  LanguageUnicode fromDisk = null;

  try
  {
    inFile = new ObjectInputStream(new FileInputStream(filename));
    fromDisk = (LanguageUnicode) inFile.readObject();
  }
  /* CATCH VARIOUS I/O EXCEPTIONS */
  return fromDisk;
} // end of GetLanguageUnicodeFromDisk
```

This method can be used instead of the constructor if desired by another class. It illustrates the simplicity of loading Java objects straight from disk. It also forms the basis for a similar method that loads a saved Java object not from the local disk but over an Internet connection by the versatile Hypertext Transport Protocol (HTTP), and coded to demonstrate the ability to use the Language component in other applications as described in Chapter 3:

```
public static LanguageUnicode getLanguageUnicodeFromWebsite(String
languageName)
{
  languageName += FILE_EXTENSION_DELIMITER + FILENAME_EXTENSION;
  URL fileURL = new URL("http://www.cus.org.uk/~alasdair/braille/" +
    languageName );
  ObjectInputStream inObject = new
    ObjectInputStream(fileURL.openStream());
  LanguageUnicode fromWeb = (LanguageUnicode) inObject.readObject();
  inObject.close();
  return fromWeb;
}
```

This demonstrates the simplicity of loading Java objects, and the way the Java streams can combine to provide easy I/O functions. In this case the `ObjectInputStream` wraps an input stream from a network connection rather than from a local file, but after this is done the mechanism to retrieve an object from it is exactly the same. The various Exceptions that can be thrown have been omitted for clarity.

The Internet location of the file is hard-coded into the class. This is convenient, in that a developer using the class would only have to know the language filename and that the class would have access to an HTTP connection to the Internet. However, it does require this location to contain a valid language file and for this to be maintained. If this approach is developed further, a resource location in the UMIST domain would be more appropriate. Alternatively, an approach could be followed that is in line with the way that the location of disk files on the local filesystem is left to the responsibility of components outside the LanguageUnicode class. The languageName parameter passed to the method could be required to be a full network location, determined by another application or record. This would make the method more flexible, but pass the responsiblity and work onto another developer of the LanguageUnicode class.

Using these classes creates the possiblity of many new Exceptions to be thrown, which are handled by throwing them back to the calling class with an explanatory note. This is the policy followed throughout LanguageUnicode. For example, in `getLanguageUnicodeFromWebsite`:

```
catch (ClassNotFoundException e)
{
  throw new ClassNotFoundException("Error parsing input stream from ″
    + ‶network.  LanguageUnicode Object not found in input stream: "
    + e);
}
```

The mechanism of saving and loading objects to disk is termed *serializing* the object. The object is required by Java to implement the `java.io.Serializable` interface, which does not require any methods to be implemented or provide any constants. Rather, it flags that this object can be serialized to a file. This task is actually performed by the `readObject` and `writeObject` methods of the various compatible `Input` or `OutputStreams`, like `ObjectInputStream` and `ObjectOutputStream`. The LanguageUnicode object is declared as serializable by implementing the `Serializable` interface. Since this interface contains no method prototypes, it essentially labels the LanguageUnicode class as one that can be serialized. This means that all of its constitutent parts - Hashtables, Strings, TranslationRuleUnicode objects - must all serializable in turn. Any parts that should not or cannot be written to disk - typically references to non-serializable objects or temporary data structures only to do with the current state of the object - can be marked as `transient` and will be ignored for serialization. The library Java classes for data structures, like Hashtable, all implement Serializable as a matter of course, so extending the serialized graph of connected objects was limited to the translation rule and wildcard classes. Again, this was simply a matter of similarly declaring them to be serializable. This is a far more complex where multi-threaded applications are being developed. A bigger problem is that serialization fixes the current state of the class quite restrictively. Serialized objects automatically obtain a unique version number. Any further development to the class, its data structures or methods, will change this number, and attempting to load in an object of differing version will fail. This requires the objects to be written to disk to be re-written every time a change is made in development. More seriously, in a release of the class no change could be made after release without breaking the old class for using newer classes. This would especially apply to LanguageUnicode objects obtained from the Internet. Older versions would have to be maintained for users of older LanguageUnicode versions. This is a disadvantage compared to the standard legacy machine format, which is independent of the progam using it. However, the storage of the Language information as a Java class is required by the complex nature of the representation of data in LanguageUnicode. Parsing a language file into a class every time would be prohibitively time-consuming and complex.

This is especially so since the move to a different file format has allowed the addition of some extra information to the language:

name

A String title for the language, e.g. "British English Braille"

description

A String for supplementary information for the language, e.g. "Standard British English Braille language, contracted and uncontracted, from RNIB 2001".

stateDescriptions

An array of Strings that contains a brief description of each state, e.g. "Uncontracted text to Braille".

inputClassDescriptions

An array of Strings that contains a brief description of each input class, e.g. "Computer Braille characters".

This additional information will allow a LanguageUnicode object to be queried using the various `get` methods and provide information on the language and what it does without referring to the human-readable original text file for the language rules table. At present the machine-format language file does not contain information on the language it encodes - its name, or what the possible states can be used for. The original human-editable file must be consulted, and the information may not all be there. This is a drawback which these additions will address. It is envisaged that these fields might be used by an implementation to present language information. This should allow an end user to make informed decisions on which state/language to use. The only other alternative is to hardcode information on known language classes and data files into a Language implementation, which is most undesirable. Simple `get` methods have been provided to allow access to the information.

If LanguageUnicode languages are stored as serialized Java objects there must have been a point when the LanguageUnicode object was constructed from the constituent data. This is the role of the LanguageUnicode version of the Make utility that turns human language files to machine format.

4.4 The Make utilities

Two more classes have been developed for creating machine-format language rules table files. One, MakeLegacy, performs an identical function to the C Mk program. It parses a language file from its human format to the compressed machine format used by BrailleTrans, LanguageInteger and Language256. MakeLegacy output files can be used by BrailleTrans. The second, MakeLanguageUnicode, parses a human format file into a LanguageUnicode serialized object. This allows the Unicode-based system to use objects as described above.

For convenience, a class Maker was developed which provided a number of common utility methods and constants. This was then subclassed by the two make-language utility classes, which inherited the methods and constants and saved recoding these functions. In both the subclasses the `make` functions have been placed in public methods. They take the full path and filename of the languages files to process and output to as parameters, just like the Language constructors. This allows the `make` functions to be reused to form part of another application. In addition, each has a `main` method to provide a simple command-line interface to the `make` functions, which parses the command line to find the relevant filenames from the user to allow simple processing:

```
public static void main(String[] args)
{
  String inputFilename = "";
  String outputFilename = "";

  // check the correct arguments have been provided
  if (args.length != 2)
  {
    System.out.println("MakeLanguageUnicode");
    System.out.println("USAGE: java brailletrans.MakeLanguageUnicode
      <from> <to>");
    System.exit(SUCCESS);
  }

  // get the pathnames and filenames from the command line
  inputFilename = args[0];
  outputFilename = args[1];

  // call the make method with the paths and filenames
  make(inputFilename, outputFilename);
} // end of main

static public void make(String inputFilename, String outputFilename)
{
  /* MAKE FUNCTIONS */
}
```

Two points are important with MakeLegacy. First, while it performs exactly the same task as Mk, it should be explicitly noted that it assumes that the human format file is encoded in a native 256-character set format. The contents are read as `ints` and parsed correspondingly. The simple `BufferedInputStream` and `BufferedOutputStream` classes are used to read and write byte stream information. These files are designed to be edited by the most simple text editors, which will typically use 256-character sets even on Unicode-based operating systems (for example, Windows Notepad on MS Windows '98). All the issues of platforms-specificity apply to MakeLegacy. For example, if a human-format language file is viewed on a platform with a different 256-character set, the contents of the file will not be the same as they were on the machine on which the language file was written. MakeLegacy will in fact produce a correct machine-version file from it, but it will correctly translate only text encoded in the same character set as the system where the language file was composed.

Second, it correctly implements an escape-character system for representing characters in the range 0 to 255 that Mk fails to. A language rules table composer may wish to include in the language rules table some characters that cannot easily be inserted by the editor. If the character is not directly accessible on the keyboard, it may be available through a combination of key-presses, or through a helper application. However, it is quite possible that the character may not be correctly represented because of the local character settings. Allowing escape characters permits the explicit and platform-independent representation of any character in the 256-character set in ASCII only. Escape characters are indicated by an initial "\x" string and followed by a decimal value in the range 0 - 255. They can be used in the translation rule table and the character mapping in the character rule table. They cannot be used in the flags or wildcards. These must all be ASCII characters, and wildcard flags must be indicated by Western Latin letters in the range A to Z, where capitals indicate the flag is set (true) and lower case that the flag is reset (false). This simplifies the implementation considerably - allowing wildcards is complicated - and since every 256-character set is a superset of ASCII it is not a serious limitation.

The MakeLanguageUnicode class must support Unicode language files, which have the ability to represent the possible characters for the language. The first is fulfilled by using a Unicode format for the language data file. This requires a Unicode-compatible editor, which is not necessarily available by default on every operating system. However, it is to be expected that the small number of producers of language tables are reasonably skilled people who can acquire a Unicode-capable editor. The Unicode variant selected is big-endian UTF-16, without an informative initial character - this is the same as internal Java Unicode representation and is also standard network byte order. This is required because the native encoding of a system is almost certainly not Unicode for text files, and the JVM will import using the native encoding by default if no other encoding is specified. The use of any other encoding mechanism must be explicitly flagged to Java when the input stream is created. MakeLanguageUnicode uses the `java.io.InputStreamReader` class. Reader/Writer classes of `java.io`, unlike InputStream/OutputStream, read and write Unicode characters rather than ints and bytes. They perform any necessary encoding or decoding. MakeLanguageUnicode therefore uses the big-endian InputStreamReader:

```
InputStreamReader fileIn = new InputStreamReader(new
   FileInputStream(filename), "UTF-16BE"));
```

This permits the normal reading of `ints` from the input stream, which can cast directly into 16-bit `char` Unicode characters and Java Strings.

The second requirement for the MakeLanguageUnicode class is that it constructs a LanguageUnicode class. To this end, a LanguageUnicode constructor without parameters, and a number of `set` methods, have been added to LanguageUnicode. Using these methods in the correct order allows data to be added to the language until the language is complete. The completed object is then serialized and written to disk.

The methods provided to add data must be accessed in a certain order. For example, a character rule cannot be provided until the number of character rules is defined. Defining the number of character rules with `setNumberCharacters(int numberCharacters)` also triggers the declaration and instantiation of the Hashtables for the language. Only then can new character rules be provided. The number of characters cannot be amended. If either of these sequence rules is broken, the set method will throw another new Exception, `LanguageDefinitionException`, to be handled by the calling class. The exception is also thrown if one of the defined language values is exceeded, for example in trying to add one too many character rules when the defined number has already been added. Whether or not a variable has been defined is tracked by initially assigning it an impossible value, defined as the constant UNDEFINED, in fact the impossible value for this context of -1. Checking for this value will indicate whether the attribute has yet been set.

```
public void addCharacterInformation(Character from,
  Character to, Integer flagValue) throws LanguageDefinitionException
{
  // check number of character has been defined
  if (numberCharacters == UNDEFINED)
    throw new LanguageDefinitionException("Number of characters must ″
    + "be set before character information can be added.  Use "
    + "setNumberCharacters");

  // check character rule limit has not been reached
  if (characterMapper.size() == numberCharacters)
    throw new LanguageDefinitionException("The number of characters "
    + "for this language has been set to " + numberCharacters + ". "
    + "limit in character rules has been met.  No more character ‶
    + ‶rules may be added");

  // add character information to Hashtables
  characterMapper.put(from, to);
  charFlagGetter.put(to, flagValue);
} // end of addCharacterInformation
```

This controlled access makes the resulting LanguageUnicode object robust and internally consistent. It does require the composer of the language table to define the number of characters supported, where this was always 256 in the legacy files. The new language attributes (name, description and so on) must also be provided. However, the number of translation rules is not required to be given in the file. Because the translation rules are stored in expandable linked lists, they can be read in at will. This saves the table author from having to keep track or count the number of translation rules.

This illustrates an attempt to make the language rule tables consistent with the old format. The same order of components, comment characters, layout are all maintained. This makes it easier to move between the two file types. Much of the file will look identical, and converting one to the other is simple. The main difference is in the character rule table, where because the number of characters is no longer fixed, the rules are now defined as:

Character to map from in input (1 character)

When found in the input text, replaced by the following.

Character mapped to in input and used for translation (1 character)

This is the result of a character mapping in the input text. It is also the key to retrieving character flags and translation rules from the Hashtable since it is the character used in translation.

Character flags (as many as required)

These are the character flags used to match left and right context wildcards. They are defined as uppercase set (true), lowercase reset (false), as defined by the Java language knowledge of Unicode. This allows the flexibility of using non-ASCII character flags, but does require that the Java specification of upper/lower case characters to be checked before they are used.

One consequence of the character rules showing only valid mappings is that invalid characters are not explicitly dealt with. As described above, invalid characters are mapped to whitespace. This will include all of the control characters, values 0 to 31, which are explicitly mapped to whitespace in 256-character language tables. (A language that doesn't use the character ‘ ‘, a space, as an indicator of formatting between words will not work properly, since the assumption that ‘ ‘ indicates whitespace is hard-coded into wildcard matching for all these languages. This limits extent to which the class can claim to be a universal translator, but not significantly. The use of ‘ ‘ is very common, and errors will only occur when searching "outside" the input text as described above.

4.5 Testing and translating utilities

Testing the efficacy and performance of the classes described above has required the development of a number of separate programs that do not form part of the project deliverables but require some explanation.

Command-line translation utilities

Each of the translation implementations required a simple command-line interface. These permit translation to be performed on test text and the results examined for accuracy. The command-line interfaces used were very similar to that detailed above for the Maker class. The file to be translated, the state to use and the language file to use can all be selected at the command line. The file was read from disk, the language initialised and run. The output was displayed on the screen, or in a Java GUI window for the LanguageUnicode Unicode output - this requires a Java 1.3-compliant JVM, but the LanguageUnicode class of course does not. A number of common functions, like reading files from disk, were coded into a `Translator` class, and this was subclassed by the various command-line utilities to allow reuse of the common code.

Output comparison test class

This utilised the Java GUI facilities again to permit text panels of the results of each of the classes to be laid out and manipulated. Differences in output, indicating inconsistent operation and therefore problems, were easier to identify.

Performance comparison test class and input files

Performance is a key concern for Java programming. A performance class ran a specified number of tests on a given translation implementation and produced results for loading and running translation, the means and standard deviations of each. A set of text files, composing files each ten times the size of the previous, produced by multiplying up or cutting down a section of this project, provided a range of input files for testing from 1 word size to 1,000,000 word size.

Existing DLL translator class

Java provides a mechanism for accessing native code, the Java Native Interface (JNI). The JVM can pass data to and receive data from native executables on the local machine. This was an opportunity to compare the Java programs against the existing BrailleTrans C program. Some amendments were required to the C BrailleTrans code to support the Java interface, but when compiled into a new DLL it could be called by a Java class. It has been used to compare speeds and output.

Dummy DLL class

Running and timing the BrailleTrans DLL through the JNI measures the performance of BrailleTrans but also the performance of the JNI. For this reason a truncated BrailleTrans DLL was produced that provided the same Java class access but performed no internal operations, returning empty values. This allows BrailleTrans performance to be differentiated from JNI performance.

Timing class

A `Stopwatch` class was created to provide simple and reusable timing functions. This allowed the other classes to simply declare a new Stopwatch, `start` and `stop` it and then obtain from it the means and standard deviations of the times. The timing mechanism available through Java uses the system time, measured in milliseconds. With the C DLL and smaller test files, this was not a high enough resolution to get precise times, and the timing results were frequently zero. Differentiating performance at this level was therefore impossible, as discussed below.

4.6 Documentation

The Java implementations of the UMIST translation system are intended to form component parts of a larger translation application. As such, they must be documented so that programmers can use the classes created in this project in their development work. This is a lesser task than producing end-user documentation for non-technical users, but it still requires the documenting of functions, behaviour, resources, rationale, public methods and attributes, as demonstrated in the Sun Java API documentation.

Java provides a simple way to document the public interface of a class or package. It is normal and good practice for program source code to be annotated with informative comments, for the benefit of future workers on the code. Java sourcecode comments that comply with a structured specification can be extracted by the **Javadoc** tool, part of the Sun Java development kit, and are automatically combined into sets of HTML webpages documenting the API. These are of the same format as the standard Sun API documentation, so they are familiar and easy to use for other Java programmers. The comment requirements are minimal. For example, the Maker utility function `turnIntoByteArray` is documented thus:

```
/**
 * Returns a byte array constructed from the array of ints passed it.
 *
 * @param toConvert    An int[] to be converted to a byte[].
 * @return toReturn    The byte[] resulting from conversion of the
 *                     int[] input.
 */
static byte[] turnIntoByteArray(int[] toConvert)
{
  /* METHOD CODE HERE */
}
```

Similar comments are applied to classes, interfaces and public attributes. The source code for all these elements can be fed straight into Javadoc, and cross-referenced structured documents for the classes and their contents is generated without further intervention. This method has been used to produce programming documentation for all the translation and language-making classes. The class documentation has been placed online for reference (kin2001).

It is hoped, however, that the classes developed in this project will themselves be developed further by future UMIST workers. This will require these workers to understand the source code and approach taken. This document will be of assistance, of course, but the source code itself is the crucial resource. Two things have been done to help future developers. The Sun-standard style was adopted for capitalisation and format of constants, methods and classes. This style is consistent throughout the classes produced to allow for easier future maintenance and work on the classes. For example, a variable is always in lower case, (`filename`), a constant in capitals (`MAX_COMPRESS`, a class with an initial capital letter (`Language256`) and so on. Second, assertions and notes have been provided throughout the source code. These were vital in understanding the original BrailleTrans code, in developing the new implementations, and for future workers on the code.

Two more pieces of documentation have been produced. One is a guide to using the command-line interface for MakeLegacy and MakeLanguageUnicode. This will assist composers of language rules tables. The second is a guide to the format requirements of the human-editable forms of the language rules tables, and their various filename extensions. This will be useful both for language composers and for future developers of these Java classes.

4.7 Packaging

The Java library uses packages to order classes as discussed above. Good Java practice requires similarly packaging newly-developed classes into related packages. This structures the class space, and makes distribution and re-use of classes far more feasible. The Sun conventions for packages were used, so the lower-case package name was determined to be `uk.ac.umist.co.brailletrans.*`. This is the network URL of the UMIST Computation Department, reversed, so the classes can be identified by future developers by their origin. The classes were split as follows:

Class	Package
TranslationLanguage, Translation256, TranslationInt, TranslationUnicode	uk.ac.umist.co.brailletrans.*
Maker, MakeLegacy, MakeUnicodeLanguage	uk.ac.umist.co.brailletrans.utils.*
Translator, timers, speed and performance tests, development classes	uk.ac.umist.co.brailletrans.tests.*

The key package for distribution and future implementation is therefore the `uk.ac.umist.co.brailletrans.*` package. A Java class that `imports` this package will have all the translation facilities of LanguageInteger, Language256 and LanguageUnicode detailed above available to it.

4.2 Performance criteria

The Java implementation of the translation mechanism will assessed on the following criteria:

- Produces the same output from the same input as the existing implementation with the same language table.
- Produces valid output using a devised Unicode-based language table.
- Performs at a comparable level to the translation rate of the existing native code implementation.

5 Results of implementations

The classes were tested to ensure that they meet their specifications (do what they are supposed to do) and perform well enough to be useful.

5.1 Validation: meeting specification

During the first stages of development the output of translation was simply displayed on the console. This allowed for visual examination of the results, which needed to be cross-checked against the language tables to confirm their accuracy. Much of the translation mechanism - left and right context checking, state machine operations and, most complex, wildcard comparisons - only applies to the production of contracted Braille. Test input text, using a number of example contractions, was devised by examining the language rules tables for a range of these contractions. The simultaneous development of three different implementations was of great value. They all applied different solutions to the problem of implementing the UMIST translation system. Discrepancies between the output when the same input text was provided to each implementation therefore highlighted an error in at least one of the implementations. Most errors occurred in the implementation of the wildcard checking in the left and right contexts, the code with the most complex logic. The existing C JNI-accessed BrailleTrans program, and the BrailleTrans port LanguageInteger, were of considerable help. Both represented an independently-coded (and presumably tested) solution, the results of which often identified errors in the LanguageUnicode and Language256 classes.

However, the existing solution is not perfect. Some discrepancies in output were identified as being the result of incorrect use of the language rules tables, not by the Java implementations, but by C BrailleTrans. As an example, the word "`SHOW`" translates in Standard British Braille, under contraction (state 2), to "`%[`", when performed by the Java classes. This can be verified by processing through the language tables manually, with initial state 2:

Correct translation process of "SHOW"

Input text	Output text	Operation
"SHOW"	""	Before translation.
"SHOW"	""	First rule with matching focus: "2 ![S]TH=S -" Fails to match - "TH" does not follow the initial "S".
"SHOW"	""	Second rule with matching focus: "2 ~[SH]'=% -" Fails to match - "'" does not follow the initial "SH".
"SHOW"	""	Third rule with matching focus: "2 ~[SH]~=SH -" Fails to match - "~" requires a space or puctuation character after the initial "SH".
"SHOW"	""	Fourth rule with matching focus: "2 [SH]=% -" Matches.
"OW"	"%"	First rule with matching focus: "2 [OW]=[- " Matches.
""	"%["	Final result.

The BrailleTrans C program does not produce this output. Without examining the C code in depth, the reasons for this cannot be ascertained. This C code is very complex, and the author's C skills are not sufficient to locate the reason for this discrepancy in output within the timescales allowed in this project. It is possible that the amendments to BrailleTrans to provide it with the same functionality as the Language classes are the cause of the apparent errors. They may stem from the project's addition of the JNI access points, JNI itself, or the addition of the variable state functionality. LanguageInteger is consistent with the other Language implementations, despite being a port of BrailleTrans, but the addition of boundary checking, variable states and other changes make it an unfaithful copy. It suffices to observe that the output of BrailleTrans cannot be regarded as entirely accurate. This is in addition to the issue of left and right context matching at the beginning and end of the input text already described. The conclusion of this untrustworthiness is that consistency with the BrailleTrans output is not sufficient as a measure of accuracy for the three Java implementations of the UMIST translation system.

A number of test input texts were devised for use. For example, "THE THE THE" tests the translation logic at the beginning and end of the input text. Only if the characters ‘outside’ the input text are assumed to be whitespace characters, the output will be "! ! !". While this kind of specific test was effective in identifying errors and discrepancies, it was not sufficient to allow the implementations to be declared to be correct and accurate. They could only be regarded as not yet being demonstrated to be in error. With very large numbers of translation rules, ensuring confidence through exhaustive testing of existing language rules tables would take a prohibitive amount of time.

An elegant solution to this problem was developed, in the form of a test language and accompanying input text. Rather than pull representative examples from an existing language, a whole new tiny language rules table was devised. It contains within it sufficient characters, states and rules to test all the logical operations of a supposed implementation of the UMIST translation system. Combined with an input text file that tests all of these components, it can standardise validation testing for all the implementations. The results of operation can also diagnose specific failures in the translation implementation. Strict ASCII characters are used for both Unicode and 256-character set versions to maximise portability (since the vast majority of systems use consistent ASCII values from 32 to 127). The language bears no relation to a human Braille or text language. Devising such a relationship is the preserve of the deviser of the language rules table. The implementations are concerned only with how accurately the rules described there are followed, not if they accurately model a text and Braille translation. The complete language is provided in Appendix Y. The logical tests fall into groups that aim to provide a comprehensive test suite when combined:

Basic mapping and translation

These are tests for the initial mapping and normalisation of the input text, and the basic mechanism of matching focus and providing output. The mappings map characters to themselves or different characters (e.g. "`A`" to "`A`" and "`B`" to "`C`"). The translation rules are very simple, taking a universal input class and empty left and right contexts (e.g. "`1 [D]=E -`". Foci of greater than one character are mapped to smaller outputs (e.g. "`1 [GG]=H -`"). One character, mapped correctly, is not provided with any matching focus translation rule ("`J`"). Errors here reflect problems with the most basic functions of the translation implementation. They should be addressed as a priority.

Left and right contexts and wildcards

These test the three wildcard classes and left and right context matching. Six characters form the foci for rules that match zero or more, one, or one or more wildcards, to the left and right (e.g. "`1 A![Q]=@ -`" where "`!`" is zero or more characters of flag type P, for which character "`1`" is provided. Failure to match "`Q`" produces "`X`" in the output. If the input text contains "`AQ A1Q A11Q`", every one should match and produce "`A@ A1@ A11@`" in the output. However, if the wildcard was to match one character only, only one of the group should match, and the output should be "`AX A1@ A11X`". Multiple characters of type P are created to ensure they all treated equally, and non-P characters are used to ensure that flag matching correctly discriminates the correct character types. Errors are most likely here, since this is the most complicated logic in the program.

State tests

A simple state table allows testing of the input class checking. One translation rule is only accessible in a state different from the default, and this state can only be obtained by the matching of another rule. The state machine can be tested by attempting and failing to match the first rule, matching the second, and then attempting and succeeding to match the first rule with the new state.

End of input text translation

As described, how the matching of wildcards and left contexts into the area outside the input text will be handled in an implementation-dependent way. Whatever way is chosen, these tests will demonstrate whether it has been implemented correctly. Characters that utilise whitespace wildcard matching in left or right context are placed at the beginning and end of the input text. This will also identify any boundary-checking failures.

The test system and specific examples were all run on a number of operating systems and platforms: the Microsoft DOS-based Windows '95 and '98 operating systems, the Microsoft Windows NT4 operating system, and Red Hat Linux (details in Appendix X). Output between the different classes was consistent, with one exception. Because LanguageUnicode processes `String` data, it does not concern itself with newlines, carriage returns and other formatting characters to the same degree as the byte-based LanguageInteger and Language256 classes. Many of the formatting characters will be lost in acquiring the input text - for example, the carriage return and linefeed used for end-of-line in a Windows text file would already have been reduced to one newline Unicode character by the process of loading the text from disk and converting it into a `String`. The two characters will still exist in the array of integers fed into the other classes. Whitespace is therefore sometimes reduced between characters in LanguageUnicode output where newlines occurred, compared to LanguageInteger and Language256 output. Since text formatting is explicitly not the responsibility of the UMIST translation system, this is not relevant to the validation of the classes and their output. Rather, the consistent results across the test languages, particular examples from existing languages, and different operating systems, allows the Java implementations to be declared to be highly accurate. Without concrete mathematical proofs, claims of complete accuracy are impossible, but confidence in the classes is high. They are accurate enough to warrant their use as component parts of larger translation applications.

5.2 Verification: performance

This project has built a number of functioning Java implementations of the UMIST translation system. They are designed as components to be used in a larger translation application. Their performance is therefore important. If they are slow, human end-users will not be happy to use applications built around them. In a multi-tasking environment like a server, slow components will occupy system resources for longer and slow the entire system.

Performance can be regarded as the time taken to execute one of the components. It is also related to the system resources, specifically memory, taken up while the component is being executed. Other system resources, like network capacity or graphic display capacity, are not relevant since they are not used by these components (except the `LanguageUnicode` class's `getLanguageFromNetwork` method, which is a demonstration of the network-friendliness of Java rather than a core function). Inefficient memory usage is a frequent criticism of all Java programs. High memory usage will result ultimately in failure, when the JVM runs out of memory and throws an `OutOfMemoryError`, or in reduced performance. The system may be forced to use the hard disk as a virtual memory store when real physical memory is exhausted, and accessing data stored on disk is far slower than accessing physical memory. Memory shortages will require the JVM to perform garbage collection and memory reclamation more frequently. There is generally a trade-off between memory and speed - finding an optimum size for the `TranslationRule256` array is a good example. Therefore, memory problems will generally be inferred from poor speed performance where appropriate, or simply demonstrated by complete failure to execute.

Performance is meaningful only when taken as a relative measure, since ultimately speed must be fitting for purpose and need be no faster (and, again, may be more efficient in system resources if it is not). However, since the implementations are intended to be components in a larger application, it could be argued that the best speed is "as fast as possible". A future developer of a larger application will build it from components like the ones developed here. The developer will then have their own performance and optimisation work to do. The less likely a given component is to be the rate-limiting step in an application, the more welcome it is to a developer. Also, a fast component will allow more leeway in building an application perhaps containing some slower components. Overall, performance will be satisfactory, and work and developer time will be saved.

However, "fast as possible", while an end in itself, still requires some external measure to ensure that the component speed is acceptable. Comparisons in this case can be made with two

Braille translation implementations of the UMIST translation system, BrailleTrans and the Word-based translation system, and with the published speeds of other commercial translation systems. First, the real speed of the implementations must be determined.

The first comparison to make is between the three implementations, LanguageInteger, Language256 and LanguageUnicode. Overall performance can be divided into **load time**, the time taken to load a language rules table into memory, and **run time**, the time taken to process a given segment of text. Run time varies with the length of the segment of text to process, while load time does not. Load time only occurs once, with the intitialisation of a Language object. If the object is retained in the Java application using it, it can be accessed at any time to provide translation without the overhead of loading a language rules table again.

The following table shows load time in a number of situations. Exact descriptions of the environments can be found in Appendix X. It suffices here to note that the environments are all desktop machines with a range of operating systems, specifications and virtual machines. The relative performance of each implementation in a given environment is of interest:

Time taken to load in ms (ratio to LanguageInteger)			
Environment	LanguageInt	Language256	LanguageUnicode
High-spec MS-DOS, new JVM	6 (1)	17 (3)	57 (10)
High-spec MS-DOS, old JVM	13 (1)	33 (3)	206 (16)
High-spec MS Windows 2000	40 (1)	53 (1)	156 (4)
Mid-spec Linux	56 (1)	100 (2)	501 (9)
Low-spec MS-DOS	248 (1)	525 (2)	2662 (11)

LanguageInteger and Language256 are closer in performance to each other than they are to LanguageUnicode. This is to be expected, since they both use the simpler `java.io.BufferedInputStream` class to read simple stream of bytes, rather than the complex `java.io.ObjectInputStream` class to read a serialized object. Language256 is slower than LanguageInteger because parses the language rules table into the objects used to represent the language data, while LanguageInteger simply loads the data into arrays and performs far less parsing. The ratios between the implementations are very similar except for the high-specification machine, where the performances are more equal. The languages files in this case had to be loaded from a slower network drive rather than a fast local drive. Performance otherwise reflects system resources and the JVM used. Load times are under a second for all the implementations in all the environments except LanguageUnicode on the low-spec machine (2.6 seconds). This is unfortunate, since a non-Western language user is likely to both require Unicode support and also be less able to afford a high-spec machine to run the Unicode version. It is to be hoped that this load time is not, however, too long for LanguageUnicode to be used in other applications. Again, this is a one-off performance hit, taken when the language rules table is loaded. Good application design should be able to ensure that this is only performed once, or as few times as is possible.

Run time is a more complex phenomenon. A number of text files of sizes from one word to one million words were generated to test speeds over input text size. Tests were then performed on different platforms, virtual machines, compilers and so on. Full results are tabulated in Appendix 5. No significant difference was found between classes produced by the three compilation options tested (Sun's J2SDK `javac` compiler, the same compiler with the "`-O`" optimisations, and the JBuilder 3 compiler). This is to be expected with mature compilers.

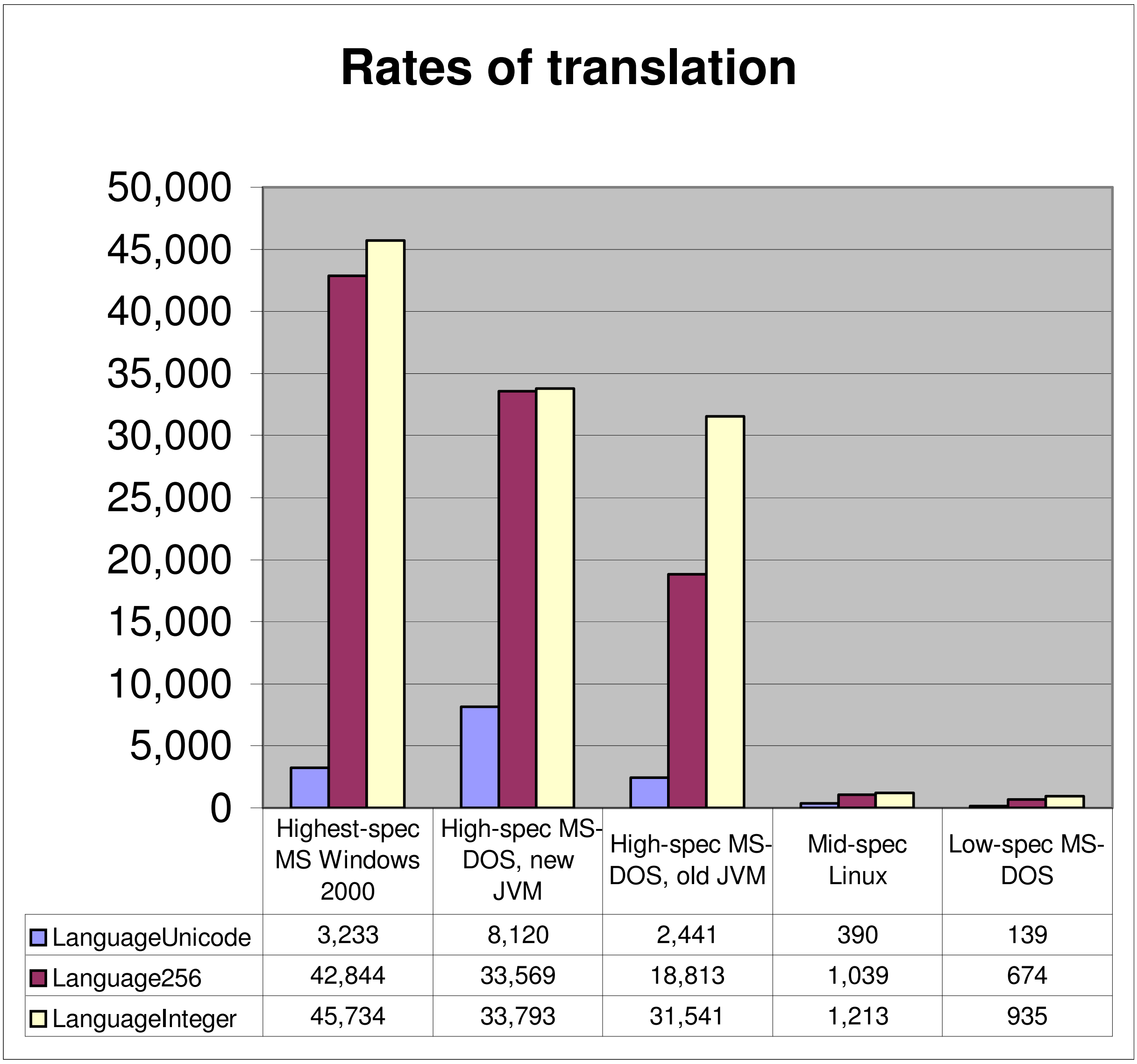
The choice of Java virtual machine was more important - later versions were faster. These factors, though, are to some extent outside the control of the coder. The environment into which the classes might be deployed is not under the control of the developer.

More interesting is how the implementations perform over a range of machines and how they can be made faster. Timed translations of smaller input files were sometimes recorded as taking zero milliseconds, so large numbers of tests were combined to produce a comparative mean. However, there were very high standard deviations relative to the mean for the small input text file sizes. A mean of zero or one millisecond might have a standard deviation of three milliseconds after ten thousand trials. This suggests that background events are more important to performance than the coding at this size of input text - whether the JVM is garbage collecting, or the screen is being refreshed, or any other quite normal call on resources. The other is that with means of close to zero and standard deviations of several milliseconds, 99% of translations are completed within 10-15ms. This is effectively instantaneous for a human observer. It means that the translation time is likely to be swamped by other tasks - like file I/O or constructing the display of output - where the component is used in a larger application. In other words, this is a very useful and successful result for those environments. Effective performance of zero time for small text segment cannot be bettered.

No problems with scalability were observed, where the processing time would increase geometrically with the size of the input text. This is not apparent, however, from the data, where rates are consistent across different file sizes up to and including hundred-thousand word input text strings (roughly the size of a short English novel). Memory problems provided the upper limits on the input text sizes that could be translated. The JVMs threw `OutOfMemoryError` errors when files of one million words were attempted except on the high-specification machines with the latest JVMs. This recommends the use of a later JVM, since memory management was a key item addressed in JVM development from most of the JVM vendors.

The following rates are therefore derived from hundreds of trials of input files of size 10 to 100,000. Files containing a single word were translated a much lower rate. This may be due to the overhead of transferring operation to the class. In any case, this will be a rare operation, and with only one word to translate the rate is not as important. The similar rates for the range 10 to 100,000 cover the likely range of input text sizes:

Rate of translation in words per second (ratio to LanguageUnicode)			
Environment	LanguageInt	Language256	LanguageUnicode
Highest-spec MS Windows 2000, mid-age JVM	45734 (14)	42844 (13)	3233 (1)
High-spec MS-DOS, newest JVM	33793 (4)	33569 (4)	8120 (1)
High-spec MS-DOS, oldest JVM	31541 (13)	18813 (8)	2441 (1)
Mid-spec Red Hat Linux	1213 (3)	1039 (3)	390 (1)
Low-spec MS-DOS	935 (7)	674 (5)	139 (1)



	Highest-spec MS Windows 2000	High-spec MS-DOS, new JVM	High-spec MS-DOS, old JVM	Mid-spec Linux	Low-spec MS-DOS
🟦 LanguageUnicode	3,233	8,120	2,441	390	139
🟥 Language256	42,844	33,569	18,813	1,039	674
🟨 LanguageInteger	45,734	33,793	31,541	1,213	935

The relative performance of the three implementations was consistent with the figures for load time. LanguageInteger ran fastest, with Language256 second. The difference between them was more pronounced in older JVMs. Higher- specification machines were better able to cope with the object-oriented design of Language256. Similarly, LanguageUnicode was slowest by a considerable margin, running at a 7-25% of the speed of the faster classes. This was especially pronounced on the older JVMs. Newer JVMs appear to be better at performing object-related operations faster – both Language256 and LanguageUnicode use objects internally to represent the language rules tables, and both benefited from the use of newer JVMs. However, as noted, the choice JVM used in operation cannot be dictated by the developer. Performance on older JVMs is important.

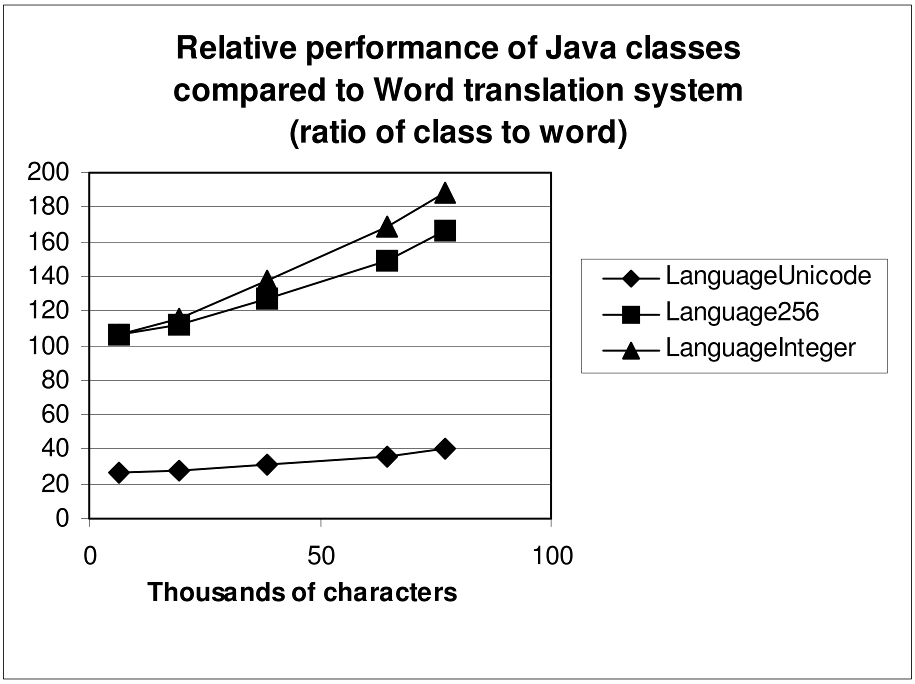
To put these relative performances into context requires data on BrailleTrans's performance. The use of the JNI to access an amended BrailleTrans DLL allowed this to be measured for the same input text files. A dummy BrailleTrans DLL allowed the relative contribution of the actual translation and the JNI overhead to be ascertained: the JNI interface contribution was too small to measure. The actual BrailleTrans results are interesting (all on the same specification machine except the worst-case):

Rate of translation in words per second					
Size of input file	1	10	100	1000	10000
BrailleTrans rate	8,264	14,085	15,723	8,873	1,354
LanguageInteger rate	5,882	25,641	33,113	34,965	36,860
Language256 rate	11,364	34,965	30,960	34,807	32,573
LanguageUnicode rate	7,042	7,905	7,899	8,535	7,893
Worst Java performance	81	138	116	131	162

These rates suggest that the C program and Java programs are as fast at the smaller input text files, but that the C program does not scale well and slows dramatically as the input file size increases. This conclusion should be drawn with caution, since the JNI is not necessarily an accurate reflection of the way the DLL would operate in a pure native environment. Access by BrailleTrans to the input text data structures passed to it may not be efficient, since it must come through the JNI. In addition, the performance of BrailleTrans on a low-spec machine could not be tested, and its comparable performance may well be better as a native application in such a constrained environment. However, at the least, these results suggests that the Java performance is not of a different magnitude entirely from the C performance, which must be regarded as a success. (The rates are also similar for the worst Java performance, that of LanguageUnicode on the low-spec machine, with those of another text to Braille translation program running on low-specification machines described in Das (1995) (das1995). The language used is not specified, unfortunately, so though the Java implementations are better than the Das system, without knowing more about the environment useful conclusions cannot be drawn).

The other key comparison possible is with the Word system. This represents a fully-featured translation application, in which BrailleTrans performs translation services. The performance of the system is therefore not a direct product of BrailleTrans's performance. However, the speeds achieved by the Java implementations can be compared to this application performance. This relative performance would indicate whether the component is fast enough to form part of such an application:

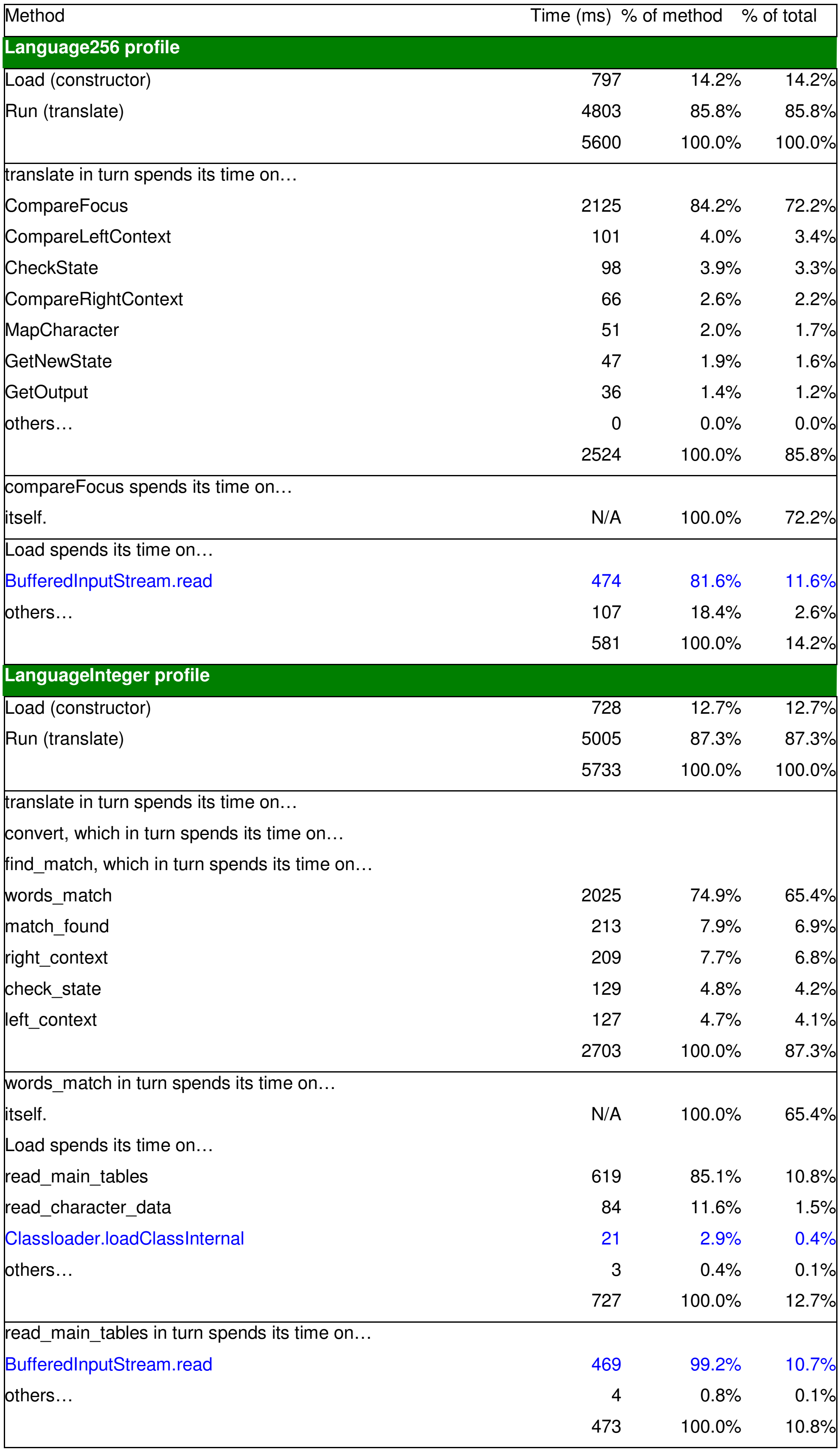
Rate of translation in characters per second (performance relative to Word system)						
Number of characters	6446	19286	38572	64460	77144	644600
Word system rate	326	306	262	229	196	N/A
LanguageUnicode	8535 (26)	8393 (27)	8106 (31)	7893 (36)	7903 (40)	8370
Language256	34807 (106)	34312 (112)	33316 (127)	32573 (149)	32616 (166)	34542
LanguageInteger	34965 (107)	35384 (116)	36230 (138)	36860 (169)	36893 (188)	38388



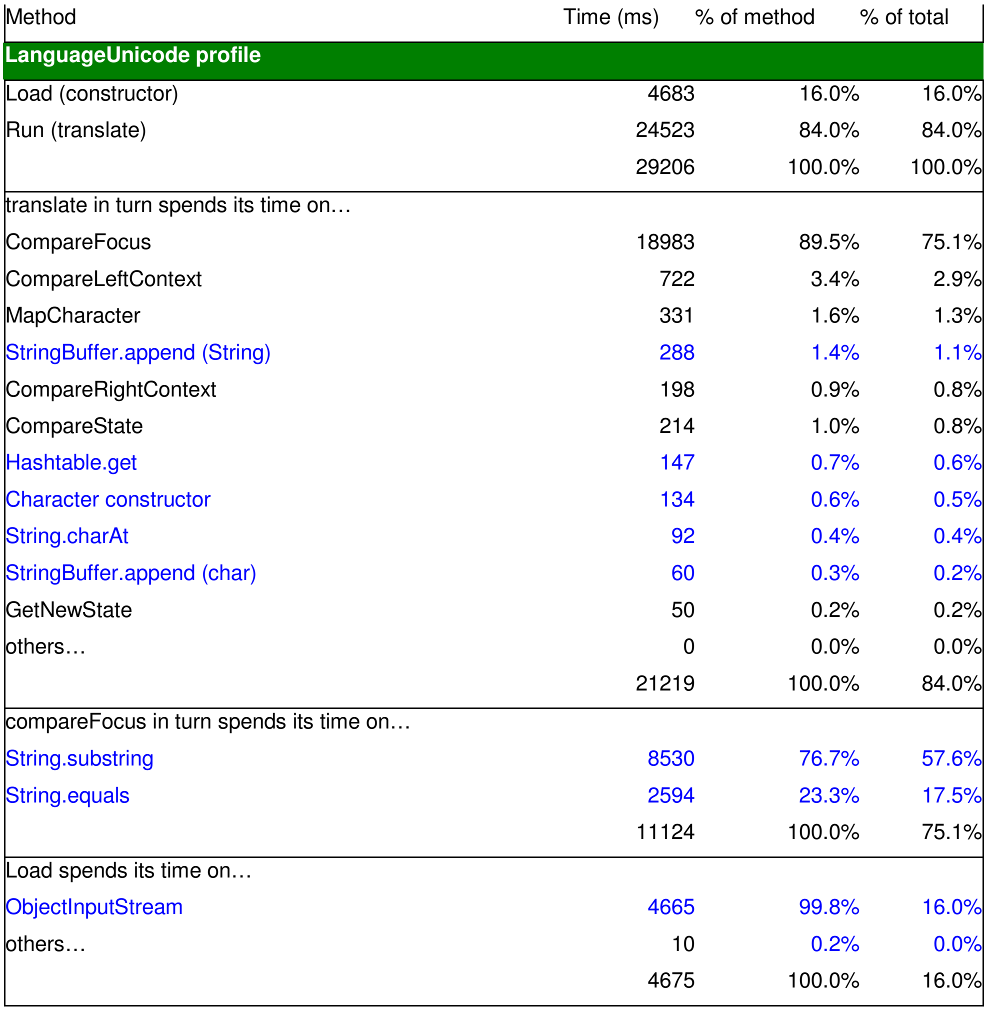
The Java classes all perform well relative to the Word translation system. The problems of the Word system in scaling to larger input text sizes, detailed in Blenkhorn, are quite clear in the better relative performance of the Java classes at higher file sizes. The Java classes do not exhibit the same scaling problems over this range of input text files. Since the limiting factor in the Word system is the rate at which text can be passed to BrailleTrans, rather than BrailleTrans itself, the Java classes would not be of assistance in resolving its performance problems. Does their performance, however, suggest that they are sufficiently fast to compose part of another larger application? The machine on which the classes were run (high-spec MS-DOS) is a faster machine than that tested for the Word system – a 700MHz processor compared to a 233MHz processor. The Java classes should therefore run considerably faster (a simple ratio according to the clock speeds is unlikely, however). The LanguageInteger and Language256 implementations, running over 100 times faster than the Word translation system, certainly appear to be fast enough even allowing for the better machine. The case of the LanguageUnicode implementation is less clear-cut. Its relative speed (twenty-six times faster than the Word system) suggests that it is in fact worthy of being considered for use. However, it is likely to best gain employ solely in situations where its Unicode functionality is vital (a non-256-character set based application) or where the platform on which it runs is guaranteed to be fast enough to negate the poorer performance of the implementation.

It is illuminating to compare this performance with that of another complete application, the Duxbury commercial translator. This is widely touted as the leading text and Braille translator. It does not have a programmer interface, so truly objective timings of performance are not possible. However, tests were run with the same input text files as used for the Java trials and the results timed by manual means. The translator achieved levels of performance better than those achieved by any of the Java classes. It handled all the input files, including the million-word input files that triggered `OutOfMemoryError` errors in the Java classes. This performance includes not only straight translation but also formatting the resulting Braille into accepted document style for Braille texts. Its performance appears to be slightly faster than the LanguageInteger implementation, but certainly no slower. This is not a poor performance for a Java program - the Duxbury program is a native executable. However, it does mean again that any future user of the Java classes in their current state must consider whether their performance is sufficient for their needs.

It may be that the performance of the Java classes can be improved by optimisation of the source code. Analysis of the three implementations was performed by use of the Java **ProfileViewer** application [dit2001]. The JVM can be instructed to dump a plain text data file with details of methods called and execution times when run. Classes were written to perform a translation operation using each of the implementations with a common input text file and the minimum of overhead outside of the translation operation itself. The resulting data files from the JVM were processed by ProfileViewer to analyse the code that is most frequently executed and uses the majority of processing time. This code is then the best candidate for any attempt to optimise the program, since any improvement will return the greatest benefit. Analysis of the three implementations produced the following results:



Method	Time (ms)	% of method	% of total
Language256 profile			
Load (constructor)	797	14.2%	14.2%
Run (translate)	4803	85.8%	85.8%
	5600	100.0%	100.0%
translate in turn spends its time on…			
CompareFocus	2125	84.2%	72.2%
CompareLeftContext	101	4.0%	3.4%
CheckState	98	3.9%	3.3%
CompareRightContext	66	2.6%	2.2%
MapCharacter	51	2.0%	1.7%
GetNewState	47	1.9%	1.6%
GetOutput	36	1.4%	1.2%
others…	0	0.0%	0.0%
	2524	100.0%	85.8%
compareFocus spends its time on…			
itself.	N/A	100.0%	72.2%
Load spends its time on…			
BufferedInputStream.read	474	81.6%	11.6%
others…	107	18.4%	2.6%
	581	100.0%	14.2%
LanguageInteger profile			
Load (constructor)	728	12.7%	12.7%
Run (translate)	5005	87.3%	87.3%
	5733	100.0%	100.0%
translate in turn spends its time on…			
convert, which in turn spends its time on…			
find_match, which in turn spends its time on…			
words_match	2025	74.9%	65.4%
match_found	213	7.9%	6.9%
right_context	209	7.7%	6.8%
check_state	129	4.8%	4.2%
left_context	127	4.7%	4.1%
	2703	100.0%	87.3%
words_match in turn spends its time on…			
itself.	N/A	100.0%	65.4%
Load spends its time on…			
read_main_tables	619	85.1%	10.8%
read_character_data	84	11.6%	1.5%
Classloader.loadClassInternal	21	2.9%	0.4%
others…	3	0.4%	0.1%
	727	100.0%	12.7%
read_main_tables in turn spends its time on…			
BufferedInputStream.read	469	99.2%	10.7%
others…	4	0.8%	0.1%
	473	100.0%	10.8%



Method	Time (ms)	% of method	% of total
LanguageUnicode profile			
Load (constructor)	4683	16.0%	16.0%
Run (translate)	24523	84.0%	84.0%
	29206	100.0%	100.0%
translate in turn spends its time on…			
CompareFocus	18983	89.5%	75.1%
CompareLeftContext	722	3.4%	2.9%
MapCharacter	331	1.6%	1.3%
StringBuffer.append (String)	288	1.4%	1.1%
CompareRightContext	198	0.9%	0.8%
CompareState	214	1.0%	0.8%
Hashtable.get	147	0.7%	0.6%
Character constructor	134	0.6%	0.5%
String.charAt	92	0.4%	0.4%
StringBuffer.append (char)	60	0.3%	0.2%
GetNewState	50	0.2%	0.2%
others…	0	0.0%	0.0%
	21219	100.0%	84.0%
compareFocus in turn spends its time on…			
String.substring	8530	76.7%	57.6%
String.equals	2594	23.3%	17.5%
	11124	100.0%	75.1%
Load spends its time on…			
ObjectInputStream	4665	99.8%	16.0%
others…	10	0.2%	0.0%
	4675	100.0%	16.0%

LanguageInteger is a complicated and already highly-optimised program. The scope for improving on its performance in sourcecode is limited. It relies on its own implementations of common operations, like matching strings, and uses simple and fast data structures. The BufferedInputStream used is the only Java library class to show up as a significant contributor to processing time (10.7% of total time in this 1000-word translation) but this is a highly efficient I/O mechanism, and loading the class is a one-off cost that can be minimised by reusing the instance of LanguageInteger created. Some performance benefit might arise from using a `DataInputStream` for Java `byte` (rather than `int`) I/O, which would require internal representation of characters as `byte` types. Looking at the method analysis for the optimization candidates, `words_match` is used extensively (65.4% of total time) because in the rule matching algorithm it is the first condition checked - every rule examined will have `words_match` run against it. The method code, however, is very simple and largely uses what optimisations Java allows - for example increments in expressions, like `table[looking++]`, which is faster, but less clear than separate `table[looking]; looking++` statements:

```
private int words_match(int up_to, int[] input_txt)
{
  int start = up_to;
  do
  {
    if (up_to == input_txt.length)  // check if we've run out of input
      return(FALSE);                // text to check

    if (table[looking++] != input_txt[up_to]) // check each character
      return(FALSE);                          // of focus
    up_to++;
  } while (table[looking] != ']'); //loop until focus all been checked
  return(up_to - start);  // Success. Return the number of input text
                          // characters matched
}
```

Optimisation would be limited to introducing one more of these incrementors to replace a standalone increment statement. No other method has any great individual impact - `match-found` contributes 6.9% to performance - and they are similarly highly-optimised, so there is little purpose to working on them. The small number of objects might be reduced still further - the `ChInfo` array could be replaced by a simpler number of arrays, containing in total all the character information; the `Output` class could be replaced with global variables. There is, however, no evidence from ProfileViewer that this will increase performance.

Language256 is very similar. The same considerations apply with `BufferedInputStream` and internal representations of characters as `bytes` rather than `ints`. Again, the majority of processing time is taken up with the Language256 equivalent of `words_match`, `compareFocus` (72.2%). Again, this is a small piece of already optimised code:

```
private boolean compareFocus(int[] input, int position)
{
  //check that focus does not exceed boundary of input text
  if (position + translationRule[transRuleIndex].focus.length >
      input.length)
    return false;
  // Use a local variable for left focus length.
  int focusLength = translationRule[transRuleIndex].focus.length;
  for (int i = 0; i < focusLength; i++)
  {
    if (input[position + i] !=
        translationRule[transRuleIndex].focus[i])
      return false;
  }
  return true;
}
```

Left focus length is used as the determinant of loop termination. To save time on looking up the left focus length from the current rule object each time, a local variable is used to store this value. No potential optimisation is immediately apparent. No other method contributes a great deal to performance.

The better performance of LanguageInteger compared to Language256 suggests that the former should form the basis of future 256-character-set translation systems, and that optimisation of Language256 would not be a good use of resources given they both have identical functionality. LanguageUnicode, however, offers Unicode translation, which may make it the implementation chosen for use in a translation application. Its poor performance relative to the other Java classes suggests that some optimisation is possible. Analysis of the ProfileViewer data does not provide much scope for simple code optimisations. In loading language files, buffering is already applied to the `ObjectInputStream` by wrapping its `FileInputStream` in a `BufferedInputStream`:

```
ObjectInputStream inObject =
    new ObjectInputStream(new BufferedInputStream(new
        FileInputStream(objectFilename)));
```

This accounts for the 99.8% of the language loading time. Since it cannot be made any faster, the only alternative is to replace the mechanism for loading language data with one more similar to the 256-character set classes, using a Unicode-based language data file that is parsed as it is loaded. This extra parsing, however, may outweigh the benefits of allowing the speedier unwrapped `BufferedInputStream` to be used. It would also require substantial recoding of language rules table creation and LanguageUnicode construction. Turning to the run methods, once again `compareFocus` occupies the majority of the processing time (75.1% of total). Again, the code for this is simple:

```
private boolean compareFocus(String toCompare, int index)
{
  // check that there is enough input text left to match this focus
  if ((index + currentRule.focusLength) > toCompare.length())
    return false;
  else
  // it does, so test to see if focus matches input text
    return currentRule.focus.equals(
      toCompare.substring(index, index + currentRule.focusLength));
}
```

There is no scope for code optimising here. Looking at the breakdown for `compareFocus` reveals that the majority of the time taken by this method and by the entire program is occupied by the Java `String` operations `substring` (57.6% of total) and `equals` in this method alone. This betrays the reason for the poor comparative performance of LanguageUnicode. Its internal use of `Strings` is the source of its relative lack of speed. Java `String` operations are high-level, powerful, and slow. Optimisation guides for Java frequently advise to avoid `String` concatenation with the overloaded "+" operator, since `Strings` are immutable and new instances are created to perform addition. However, this has been avoided in LanguageUnicode for just this reason. The only addition performed is to the output text, which is a mutable, growable `StringBuffer` - used in `String` concatenation. Similarly, the extensively-used `Hashtables` might be replaced with a local solution, as described in Section 4.3. Again, however, the `Hashtable.get` method occupies only 0.6% of processing time, so there is little call to do so. If the performance of LanguageUnicode is to be improved, it must involve a fundamental change in the design. Instead of slow `String`-based representations of the language rules table, the implementation could be recoded (or a new implementation performed) to use arrays of `chars` or `ints`, in a very similar way to Language256 or LanguageUnicode (or indeed BrailleTrans). This would likely produce a faster program. Language256 would serve as a base without requiring much amendment.

This redesign of LanguageUnicode is perhaps the only optimisation or recoding suggested by the tests performed. It is not necessarily required: the class is not slow in absolute terms (7,658 words per second is a good rate for most documents). The other classes perform well. Any work on them would require re-coding, which is likely to introduce errors, which will require more development time. The limited potential improvements highlighted by analysis of the classes in action do not suggest that this work is justified. In operation, the simple methods that occupy the majority of processing time will be candidates for compilation to native code by Just-In-Time JVMs for speed. All this militates against further work for little reward on these classes.

Resources would be better directed at recoding LanguageUnicode, or better still using the components developed to create larger translation applications. Some attempts to do just this will now be described.

5.3 Language implementations

A number of implementation possibilities, outlined in Section 3.3.3, were investigated. The results were mixed. The component design of the implementations and the split into translation engine and language rules table file was not successful in every instance.

Enable a translation system to obtain language resources over a network using HTTP over TCP/IP.

This was successful, as already described in Section 4.3 above. The serialized LanguageUnicode object was simply downloaded over the network without user intervention. The behaviour of the translation component was identical, though initialisation took longer as the Language had to downloaded over a dial-up modem link. The serialized British English table used was 61Kb, so this took around twelve seconds. It would be necessary, with such a delay, for an application performing such a transaction to alert the user to this fact. Alternatively, an application might offer the opportunity to acquire new language rules table files and download updates of older ones by going online and downloading new versions. Because of the additional descriptive information added to the LanguageUnicode language rules table the object itself should provide sufficient information for a user to tell its capabilities and how to use it.

Provide translation in a web browser.

Creation of an applet and webpage that provided translation was simple, but using it was impossible. The problem lies in the separate language rules table file. An applet runs only under stringent security conditions, which prohibit the applet from accessing the local file system to access language files or connecting to a network to obtain files. This is the case even if the applet has itself been downloaded from the same network. These security conditions may be altered, but in general few systems will permit applets to access other files. Exceptions are made for other classes, and for image and sound files, but not for other files. The only possibility identified was to include in the body of a class source code all the information contained in a language rules table and load only that information into the apple when it is instantiated. The LanguageInteger class, having the simplest structure, would be most suitable for this. For example, the translation rules table might be instantiated in the constructor for the class with `private int[] table = { 2, 16, 91, 32, 39, 69, 78 ... }`. The `ChInfo` array and `Output` object should be replaced by simple arrays or packaged with the LanguageInteger class in a JAR file. This is an inelegant solution, but would provide translation in a browser. At present, the Java classes are very small - 15 Kilobytes for LanguageInteger up to 24Kb for LanguageUnicode - and the language rules table files similarly so - 23Kb for the legacy files, 68Kb for the Unicode serialized objects. This makes them excellent for applets, where they might be downloaded over a network before use. Developing an applet with the Java translation would be an excellent application if the data file issue can be resolved, and deserves further work.

Provide translation as a consumer-device application.

Consumer devices that support Java generally support Java applets, rather than stand-alone Java applications. Mechanisms to support Java applications and dynamic downloading of components as needed do exist, but they are device-specific and very recent developments. This is a fertile avenue of future development, but requires more research.

Provide translation as a service in Sun's StarOffice word processor.

The StarOffice suite was investigated. A similarly friendly macro-based interface for the user was devised, where translating the current document could be added as a menu item to one of the main menus - File or Tool would seem appropriate. This would mirror the current Word system. Linking a Java application into this front end would be more problematic. Although the StarOffice application is now owned by Sun, it has until recently been an independent product. As such, it has not been designed with Java as a prime concern, and the application is still based around the use of its own StarOffice BASIC. The Java-friendly StarOffice API and its documentation is due to be available in October 2001. Java demos are provided for some simple functions, but some effort would be required to learn enough about the StarOffice system to allow the Java translation system to be added to it. Further investigation after the October StarOffice release is warranted. StarOffice is available for UNIX, Solaris and Microsoft platforms, and is free. This makes it a good target for an application built around the Java UMIST translation system implementation, especially since the BrailleTrans and Word system covers the majority Microsoft operating system.

Provide an online translation service through a website.

Sun has attempted to push Java into the enterprise market through Java servlets, Java classes intended to be run by a Java-enabled server in much the same way as Perl scripts or C executables have been run through the Common Gateway Interface (CGI) for years, providing server-side functionality. Servlets are therefore integrated closely with the standard Java API, which means that the Language components are easily combined with a simple servlet to provide translation functionality. This is a good example of the software engineering principle of reuse of simple components to make larger applications, facilitated by the deliberate object-oriented design of the Language translation components.

Standalone Java application

One other potential application of the Java classes was identified. Systems with the Java 2 runtime environment installed have the ability to execute JAR files, or to be precise, the `main` method of a nominated class in the JAR. With a Java 2 JVM, and access to the Java Swing and `java.io.jar` classes, there is the prospect of a standalone application that can be run from the operating system in same way as a normal native executable. This neatly solves the problem of how to integrate a Java class, which requires the JVM to be started to run it, into the operating system where normal executables are run directly. It contrasts well with the normal solutions, requiring the user to run the class using their command-line JVM interface (e.g. "`java uk.ac.umist.co.brailletrans.translators.TranslationApplication`") or, like ProfileViewer, supplying a JAR or other compressed file with system batch files, which contain the same shell commands and can be executed directly. The user runs the batch file that works on their machine, if one is present. The JAR-executable evades this clumsy arrangement, but since the installation of a Java 2 runtime environment on the machine this cannot be regarded as a real solution in the same way that a browser implementation might be.

6 Conclusions and further work suggested

The development of computer text and Braille translation is justified a need for this service in the visually impaired community. A number of private translation programs exist, but their private ownership restricts open development. A translation algorithm has been developed at UMIST and implemented in a C translation program, BrailleTrans. This has been used in turn to provide translation functionality to Microsoft Word on Windows 32-bit platforms. However, a number of limitations of this implementation were identified: the platform and character-set dependency of the C implementation, its standalone function, and the difficulties of using BrailleTrans as part of a larger application.

The project therefore proposed the development of a number of new classes to improve where possible on BrailleTrans. The development of these classes was a success, and the time available was appropriate to the work required. Validation tests indicated that the classes correctly used the information in the language rules tables to perform translation as their specification required, or convert the human format language rules tables into machine format correctly. Verification tests indicate that performance of the translation classes is generally reasonable both in absolute terms and relative to the C BrailleTrans implementation. No serious performance-degrading coding errors were discovered on analysing the performance of the classes. Since Java execution times are always slower than native executables, this good performance is superficially a considerable success. However, there are some significant performance differences amongst the three classes.

LanguageInteger and Language256 both perform identical functions on identical data, but LanguageInteger runs faster. It is more complex in internal coding, but not restrictively so, and is more similar to the existing BrailleTrans code - so an understanding of the algorithms used is transferable between the programs. There may well be future development on both BrailleTrans and the Java classes. The best solution is therefore to cease development of Language256 and use LanguageInteger as the 256-character-set translator for Java. Analysis of the two classes suggests that further optimisation of the code will not provide any significant performance. A possible recoding using `bytes` to represent characters internally does not guarantee any performance improvements and would require more development work, so is not recommended. LanguageInteger has good performance and an existing cousin for reference in BrailleTrans, and can be regarded as suitable for use in the development of a future translation application. (The limited optimisation mentioned in Chapter 5 above will be incorporated into the source code provided with the project).

LanguageUnicode does not perform well, either in comparison with the other classes or absolutely. Analysis of the class suggests that this will only be improved by re-engineering the class to use arrays of `chars` or `ints` internally instead of `Strings`. Language256 can serve a purpose here, since it already uses arrays of `ints` for text. This development is suggested as necessary before the class can be used as a basis for a future translation application. It may be that a faster implementation would not be strictly necessary - this depends on the future application. However, the increase in performance is likely to be substantial - speeds at nearer Language256 levels should be attainable - and the work required relatively simple. A Unicode implementation is important for promoting universality and widespread use of the UMIST translation system. Also, because the public interface of the LanguageUnicode class is already defined by the Language interface, development of an improved LanguageUnicode implementation can continue concurrently with development of an application intended to use it - the application will simply use the older, tested version until the newer implementation is ready, then drop the faster version into the application in place of the old.

One possible amendment to the LanguageUnicode class functionality might be desirable. At present the decision of what to do with characters that cannot be translated - formatting characters or simply those outside the range supported by the language - is left to the discretion of the coder of the class that implements Language. In all the translation classes developed the policy of BrailleTrans was continued, and the characters were copied to the output untranslated. This is a less suitable solution for Unicode-based languages, where the input is less likely to easily map to a Braille character (the problem will arise in text to Braille translation since computer Braille is represented universally in NACBC). A simple `set` method might be added to the class that allows the choice of whether to replace or use the unmatched input character to be left to the translation service user, not fixed by the coder. This requirement will probably not be important until such time as a Unicode-based language is developed. Development with the classes should not be suspended until this functionality is implemented.

The last factor concerning LanguageUnicode is its slow loading time, especially in the lower-spec machine. Object input/output, while very convenient and elegant from the programmer's perspective, appears to be slower than straight byte I/O operations. This might be overcome by adopting not only Language256's internal int arrays to represent language information in LanguageUnicode but also the storage of language rules table files as compressed text information rather than serialized Java objects. Reading them as integer arrays may be faster than reading and casting objects. However, language initialisation is a one-off performance hit, and even with the lower-spec machine, performance was not so bad as to make the application unusable. This development route might be pursued if a low-spec environment, like a PDA, required a fast real-time Unicode translation service. Otherwise, this development is of low priority.

Before moving on to a discussion of what applications might be developed, the supporting material for the Language classes should described. The two language construction classes allow for the construction of language rules table files on any platform (the Unicode class allows for construction on any platform where a Unicode file can be saved). They are self-contained, with a command-line interface. Both add the ability to express character values using *escape characters* rather than having to discover how to enter non-standard characters with a standard keyboard and assume the same character set on any subsequent use of the language file. It is hoped that the legacy file-maker in particular may prove of use to users of the Language tables or indeed of BrailleTrans because of its platform- and character-set-independence. The thorough documentation for the classes, both internally in the source code and through the Javadoc API webpages produced, provide a strong foundation for future work on or with the classes. Finally, the language test files - input, language rules table file and correct output file - provide a generic test suite for validating future implementations and will be of use to any future application development with the UMIST translation system.

While the classes were successful in simply implementing the UMIST translation system, they also aimed to be so designed as to be easily used as the basis for larger application systems. This is partly accomplished by using Java, which is platform-independent and object-oriented, both important software engineering factors. However, the design of the components and their relation to each other is vital. The use of the Language interface to define translation components structured the three implementations well, and should similarly structure future implementations if its use is continued. This aids consistency and re-use of the components by defining a common public interface to their translation functionality. Investigation of the use of the components in applications more complex than simple command-line-driven translation programs met with mixed success. The simple network functionality - downloading language rules tables over the Internet - was a good demonstration of the network focus of Java and its applications. Production of a translation servlet for server-side use was straightforward and justifies the component design well. Generally, use of the components in testing and development applications indicated that they were easily integrated into other Java classes, which is a promising sign.

However, the separation of the Java classes that provided the translation functionality from the language rules tables that provided the data for translation caused problems. This requires a Java application to have access to non-Java components, typically through the file system. Stepping outside the JVM domain in this way produces performance and security issues. These were manifested in the relatively slow disk access times for loading the data and in the difficulty of building a Java applet that could perform translation. In effect, the Language components produced are not complete. They cannot be activated as objects, whether downloaded from a network or extracted from a JAR file, and provide their functionality immediately. This restricts their ability to be used as simple building blocks for larger applications.

A further refinement is required. Language objects should be available which provide translation on instantiation without further resources or access outside of the JVM. This should be provided by use of the *Beans* component architecture for Java (sun2001g). This requires Java objects that declare themselves to be Beans to conform to a set of requirements that allow them to be loaded and run dynamically. For example, all Beans must be `serializable` and provide to external objects information about the methods and attributes of the class to which the Bean belongs. More particularly, the Beans architecture allows objects to be declared from serialized Java objects. This would allow complete Languages objects, with their internal data for translation, to be treated exactly as though they were simple classes. They could be included in JAR files and the JVM would load them into memory without requiring casting or parsing. The amount of development work required is reasonably minimal. A method to serialize the current object should be added to every Language, and a default constructor. Appropriate classes would have to be written to utilise the new Beans, but this project did not deliver any complete larger applications in any case, so that work still needed to be done. Because Beans only require the older and thus more widely supported Java 1.1 class libraries and JVMs, they can be used on a majority of platforms. The Beans can be used as components in Java GUI development environments, or applets, or any other application, elegant and self-contained.

The classes as they are, however, are well-defined components, and in any situation where the requirement for disk or network access permission is met they can be used for development of future applications. For example, the integration of translation services into the new Sun StarOffice would provide a platform-independent equivalent of the current Word and BrailleTrans system in a completely free office software environment. A standalone Java GUI application, similar to the Duxbury translation program, could be constructed with the Java Swing GUI components and still be platform-independent, though it would then be limited to Java 2 virtual machines. The older Java 1.1 Abstract Windowing Toolkit could be used for a more limited application. Any of these examples could include options to download updated language files from UMIST as part of the application. These applications could be packaged into a JAR to form an effective pseudo-executable for users with a Java 2 runtime environment, or helper batch files could be packaged with the applications. There is however perhaps more justification for developing the StarOffice application, since it would parallel the Word development and fit the current UMIST strategy. StarOffice has a high profile amongst much of the programmer community, which might benefit UMIST, and is being promoted by Sun, which means that the Java development should be able to draw upon comprehensive knowledge resources.

Other work can always be found. The Java Remote Method Invocation (RMI) (sun2001h) system allows remote objects to be accessed as though they were on the local JVM. It could potentially build on the existing ability to download language rules tables, using classes running on a UMIST server. Any server development of the classes would require some consideration of their threading characteristics, and a new Language class would need to be developed to handle UTF-8 HTTP-transported text, or a front-end to an online system provided to upload text files in an appropriate encoding mechanism.

However, these considerations are low priority possibilities. They are solutions to problems that have not yet been encountered. A more important potential development is a producing an application for a limited consumer device. Several devices exist that can run Java applications of Java 1.1. The user interface would therefore take up most of the time, and development could begin immediately with LanguageInteger as the active component. The benefits of opening up translation services through the UMIST system to a new segment of users are considerable, and this development is highly recommended.

A final proposal is for an application that although unexciting might be of more value than an expansion of the translation functionality. The UMIST translation system relies upon the construction of language rules tables for each and every language that is to be supported. The production of these tables involves composing and editing large text files, with fixed layouts and hundreds of items of information. A Java class or classes that places an easy-to-use, possibly graphical front-end onto these language files, and performs checks on the file to ensure that the information required is present and in the correct format, would greatly simplify the task of constructing such a language. Encouraging the production of languages - especially now a Unicode-based translator is available - would be a self-contained and straightforward benefit for the translation system and its users.

Appendix 1 - Computer Braille Code

Also known as "North American Braille Computer Code". This table is reproduced from the website of the Keio University *Access Research Group* [kei2000].

dot6	dot5	dot4	dot3	dot2	dot1	Character	ASCII value in decimal	Braille character
0	0	0	0	0	0	SPACE	32	
0	0	0	0	0	1	A	65	⠁
0	0	0	0	1	0	1	49	⠂
0	0	0	0	1	1	B	66	⠃
0	0	0	1	0	0	'	39	⠄
0	0	0	1	0	1	K	75	⠅
0	0	0	1	1	0	2	50	⠆
0	0	0	1	1	1	L	76	⠇
0	0	1	0	0	0	@	64	⠈
0	0	1	0	0	1	C	67	⠉
0	0	1	0	1	0	I	73	⠊
0	0	1	0	1	1	F	70	⠋
0	0	1	1	0	0	/	47	⠌
0	0	1	1	0	1	M	77	⠍
0	0	1	1	1	0	S	83	⠎
0	0	1	1	1	1	P	80	⠏
0	1	0	0	0	0	"	34	⠐
0	1	0	0	0	1	E	69	⠑
0	1	0	0	1	0	3	51	⠒
0	1	0	0	1	1	H	72	⠓
0	1	0	1	0	0	9	57	⠔
0	1	0	1	0	1	O	79	⠕
0	1	0	1	1	0	6	54	⠖
0	1	0	1	1	1	R	82	⠗
0	1	1	0	0	0	^	94	⠘
0	1	1	0	0	1	D	68	⠙
0	1	1	0	1	0	J	74	⠚

dot6	dot5	dot4	dot3	dot2	dot1	Character	ASCII value in decimal	Braille character
0	1	1	0	1	1	G	71	⠛
0	1	1	1	0	0	>	62	⠜
0	1	1	1	0	1	N	78	⠝
0	1	1	1	1	0	T	84	⠞
0	1	1	1	1	1	Q	81	⠟
1	0	0	0	0	0	,	44	⠠
1	0	0	0	0	1	*	42	⠡
1	0	0	0	1	0	5	53	⠢
1	0	0	0	1	1	<	60	⠣
1	0	0	1	0	0	-	45	⠤
1	0	0	1	0	1	U	85	⠥
1	0	0	1	1	0	8	56	⠦
1	0	0	1	1	1	V	86	⠧
1	0	1	0	0	0	.	46	⠨
1	0	1	0	0	1	%	37	⠩
1	0	1	0	1	0	[	91	⠪
1	0	1	0	1	1	$	36	⠫
1	0	1	1	0	0	+	43	⠬
1	0	1	1	0	1	X	88	⠭
1	0	1	1	1	0	!	33	⠮
1	0	1	1	1	1	&	38	⠯
1	1	0	0	0	0	;	59	⠰
1	1	0	0	0	1	:	58	⠱
1	1	0	0	1	0	4	52	⠲
1	1	0	0	1	1	\	92	⠳
1	1	0	1	0	0	0	48	⠴
1	1	0	1	0	1	Z	90	⠵
1	1	0	1	1	0	7	55	⠶
1	1	0	1	1	1	(	40	⠷
1	1	1	0	0	0	_	95	⠸
1	1	1	0	0	1	?	63	⠹

dot6	dot5	dot4	dot3	dot2	dot1	Character	ASCII value in decimal	Braille character
1	1	1	0	1	0	W	87	⠺
1	1	1	0	1	1	]	93	⠻
1	1	1	1	0	0	#	35	⠼
1	1	1	1	0	1	Y	89	⠽
1	1	1	1	1	0	)	41	⠾
1	1	1	1	1	1	=	61	⠿

Appendix 2 - Existing language rules table

These are excerpts from the Standard British English text to Braille language rules table, unprocessed, to demonstrate the editable text format of the files.

Character rules

These are in the format:

1. ASCII Value of the character to be translated:
2. Character to use in the normalising process
3. Character as an upper-case version (not used)
4. Character characteristic flags. These show
    1. text character
    2. digits
    3. punctuation
    4. white space character
    5. possible roman letters
    6. not used
    7. wildcard character
    8. Capital signs

Upper case indicates flag is set.

```
032:  :ldpSrsWc
033:!!:ldPsrsWc
034:"":ldPsrswc
048:00:lDpsrswc
058:::ldPsrsWc
065:AA:Ldpsrswc
066:BB:Ldpsrswc
091:{[:ldPsrswc
```

Wildcard specification

The wild cards are characters already included in the character translation table. The character and their minimum number are given, followed by the symbol ‘.’ for a single instance and ‘+’

for multiple instances. Then comes the decimal value of the bit(s) to be set in the character flag of the wildcard, corresponding to the flags of the character translation table.

The whole is prefixed by the number of wild cards in total.

```
7
*!=1+ 1
*#=1. 2
*~=1+ 12
* =1. 8
*|=0+ 128
*:=1+ 16
*;=0+ 1
```

Decision table

The decision table is a n times m matrix of boolean values, where n = number of machine states and m = possible input classes. These will be language-specific: the decision table can be as large as needed to accommodate the role the finite state machine plays in translation. In English to Standard English Braille (Grade I or II) a 5 (state) times 6 (input classes) table is required.

The table is presented as:

- The number of states (n)
- The number of input classes (m)
- The decision table, n lines with m digits. A non-zero value indicates a true result, indicating that a rule may be used. The input class number is used as the non-zero, which enhances readability.

The English to English Braille file reads therefore:

```
5
6
103456
120006
103006
100406
000056
```

Translation rules

These are in the format:

1. Input class
2. Translation rule: left context [focus] right context = output
3. New state

```
2       ~[HM]~=H'M     -
1        [H].~=;H      -
1       [H]=H   -
5       [H]=H   -
```

The end of the rules is delineated by a line with a single '#' character.

Appendix 3 - Test results

A number of test files were used for translation. They were composed of sections of this project write-up, scaled up to large file sizes to allow the maximum capacity of the classes to be determined.

Test file sizes		
	Words	Characters
size0	1	8
size1	10	57
size2	100	671
size3	1,000	6,446
size4	10,000	64,460
size5	100,000	644,600
size6	1,000,000	6,446,000

To confirm the platform-independence of the classes and to measure performance on different machines, a number of different computers were used for testing. Details are provided here.

Test machines			
Name	Processor	Memory	Operating System
“Highest-spec”	Intel x86 866MHz	262Mb	Microsoft Windows 2000 5.0 Build 2195 SP2
"High-spec"	AMD Duron 700MHz	384Mb	Microsoft Windows '98 4.10.2222 A
"Medium-spec" (Linux)	Intel Pentium 2-MMX 333MHz	128Mb	RedHat Linux 5.1
"Low-spec"	Intel 486DX-2 66MHz	16Mb	Microsoft Windows '95 4.00.950a

A number of different JVMs were used:

Virtual machines	
Name	Used on
Sun JVM 1.3.1	"High-spec"
Sun 1.1.8	"High-spec"
Sun 1.2.2	"Low-spec"
Microsoft VM 5.00.3802	"High-spec"
Sun 1.2.2	"Medium-spec" (Linux)
Sun 1.2.2	"Medum-spec" (Windows NT)

Finally, the test language input and output files:

Translation testing		
Input	Tests	Correct output
YY	Whitespace outside input text logic	@X
A B D E GG J	Character mapping and focus matching	A C E G H K
XXXXX XXXX XXX XX X	Correct movement through input text	5 4 3 2 1
AQ A1Q A11Q LA L1A L11A	"Zero or more" wildcard	A@ A1@ A11@ @A @1A @11A
AR 2R 3R M2 M3 MA	"One only" wildcard	AX 2@ 3@ @2 @3 XA
AS A4S A3S A44S NA N4A N3A N44A	"One or more" wildcard	AX A4@ A3X A44@ XA @4A X3A @44A
T V T W	State machine logic	T @ U V
ZZ	Whitespace outside input text logic	X@

Finally, the full test results. Some notes are required:

- OOME is an abbreviation for `OutOfMemoryError`, and indicates that the tests were not completed.
- Different testing procedures were used for larger input files – notably the collection of standard deviation information. Highlighted cells indicate actual results.
- Contracted Braille was faster than uncontracted, reflecting fewer character translations. Grade II contracted British English text to Braille (britishtobr) was used for all of the tests.
- The three compilers tested were Sun SDK 1.3 (javac), same compiler with optimisation enabled (javac –O) and the Borland JBuilder 3 (jbuild3) compiler.
- Mean performances do not take into account the results of translating one-word input files, since these were depressed by the general overhead of starting up the translation process.

Bibliography

[bik2000] Bik, Aart J.C., Girkar, Milind & Haghighat, Mohammed R.

"Experiences with Java JIT Optimization", Java JIT Compiler References website, July 2000. <http://www.eecg.utoronto.ca/~brewste/java/jit/intel-jit-experiences.pdf>

[bel1997] Bell, Doug

“Make Java fast: Optimize!”, JavaWorld website, April 1997, <http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html>

[ble1995] Blenkhorn, Paul

"A System for Converting Braille into Print", *IEEE Transactions on Rehabilitation Engineering*, Vol 3, No 2, pp 215 – 221, June 1995.

[ble1997] Blenkhorn, Paul

"A System for Converting Print into Braille", *IEEE Transactions on Rehabilitation Engineering*, Vol 5, No 2, pp121 - 129, June 1997.

[ble2001] Blenkhorn, Paul & Evans, Gareth

"Automated Braille Production from Word-Processed Documents", *IEEE Transactions on Rehabilitation Engineering* Vol 9, No 1, pp 81 - 85March 2001.

[brl1997] BRL: Braille through remote learning

"Braille Music Code", BRL website, 1997, <http://www.brl.org/music/index.html>

[bra1999] Braille 2000 Working Party

“Braille 2000: Meeting the challenges of a new millennium”, Blind Citizens Australia website, March 1999, <http://www.bca.org.au/brl2000.htm>

[bru1991] Bruce, I., McKennell, A. & Walker, E.

“Blind and partially-sighted adults in Britain: the RNIB survey”, Volume 1, H.M.S.O, London, 1991.

[cra1997] Cramer, Timothy, Friedman, Richard, Miller, Terrence, Seberger, David, Wilson, Robert, & Wolczko, Mario

"Compiling Java just in time", *IEEE Micro* Vol 17 No 3, May/June 1997. <http://dlib.computer.org/dynaweb/mi/mi1997/@ebt-link;hf=0?target=if(eq(query('%3cFNO%3e+cont+m3036'),0),1,ancestor(ARTICLE,query('%3cFNO%3e+cont+m3036')))>

[das1995] Das, Pradip K., Das, Rina & Chaudhuri, Atai

"A Computerised Braille Transcriptor for the Visually Handicapped", *Proceedings RC IEEE-EMBS & 14th BMESI*, pp 3.7 – 3.8, 1995.

[dav2001] Davis, Mark

“Unicode Newline Guidelines”, Unicode, Incorporated website, March 2001, < http://www.unicode.org/unicode/reports/tr13/index.html>

[dit2001] Dittmer, Ulf & White, Greg

“ProfileViewer”, application website, April 2001, <http://www.capital.net/~dittmer/profileviewer/index.html>

[dur1991] Durre, Karl P., Tuttle, Dean W. & Durre, Ingeborg

"A Universal Computer Braille Code For Literary And Scientific Texts", *International Technology Conference*, December 1991. Unpublished paper. <http://www.Braille.org/papers/unive/unive.html>

[dur1996] Durre, I. K.

“How much space does Grade 2 Braille Really Save?”, *Journal of VisualImpairment and Blindness*, pp 247 – 251, Vol 90(3), May-Jun 1996.

[dux2001] Duxbury Systems

April 2001, <http://www.duxburysystems.com>

[eck2001] Eckel, Bruce

“Comparing C++ and Java”, JavaCoffeeBreak website, June 2001, <http://www.javacoffeebreak.com/articles/thinkinginjava/comparingc++andjava.html>

[fow1997] Fowles, Ken

“Character sets”, Microsoft websites, June 1997, <http://www.microsoft.com/typography/unicode/cs.htm>

[gal2001] Galli, Peter

“Java to overtake C/C++ in 2002”, ZDNet website, August 2001, <http://www.zdnet.com/eweek/stories/general/0,11011,2804967,00.html>

[gos2000] Gosling, James; Joy, Bill; Steele, Guy & Bracha, Gilad

"The Java Language Specification, Second Edition", Sun Microsystems website, July 2000. <http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html>

[gsc1994] Gschwind, Mike

"ISO 8859-1 National Character Set FAQ", Verein der Informatik Studierenden website, 1994, <http://www.vis.ethz.ch/manuals/charsets/FAQ-ISO-8859-1.html>

[hag2000] Haggar, Peter

“Use the Java libraries judiciously”, IBM Developers’ website, 2000, <http://www.developer.ibm.com/library/articles/programmer/haggar_use.html>

[hed1998] Hedánek, Jirí

“An introduction to the pronunciation of Czech”, Charles University of Prague website, 1998, http://www.ff.cuni.cz/departments/fu/jh/jh-czpro.html

[hms2000] Her Majesty’s Stationery Office

"The Telecommunications (Services for Disabled Persons) Regulations 2000", Her Majesty's Stationery Office, HMSO website, October 2000, <http://www.hmso.gov.uk/si/si2000/20002410.htm>

[hot2001] HotBraille

March 2001, <http://www.hotBraille.com>

[irw1955] Irwin, Robert B

"As I saw It", pp 3 -56, quoted on the New York Institute for Special Education website, <http://www.nyise.org/blind/irwin2.htm>, September 2001.

[kei2000] Keio University Access Research Group

"NABCC (North American Braille Computer Code", Keio University ARG website, January 2000, <http://buri.sfc.keio.ac.jp/access/arc/nabcc.html>

[kin2001] King, Alasdair

"Braille Language API documentation", personal website, September 2001, <http://www.cus.org.uk/~alasdair/2001/braille/javadocs/index.html>

[lor1996a] Lorimer, Pamela

"Researches carried out in endeavours to make the Braille code easier to read and to write", Internation Braille Research Center website, December 1996, http://www.braille.org/papers/lorimer/chap6.html

[lor1996b] Lorimer, Pamela

"Researches carried out in endeavours to make the Braille code easier to read and to write", Internation Braille Research Center website, December 1996, <http://www.braille.org/papers/lorimer/chap8.html>

[mca1996] McCall, Steve & McLinden, Mike

"Towards an inclusive model of literacy for people who are blind and who have additional learning difficulties", *The British Journal of Visual Impairment,* Vol 15(3), 1996.

[mic2000] Microsoft Corporation

"C# Introduction and overview", Microsoft website, June 2000, <http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp>

[mic2001] Microsoft Corporation

Microsoft Visual Basic homepage, Microsoft website, September 2001, <http://msdn.microsoft.com/vbasic/>

[obe1990] Oberleitner, W., & Seiler, F. P.

"WineTU: German language Grade 2 to ASCII braille translator", *Journal of Microcomputer Applications*, Vol 13, pp 185 – 191, 1990.

[omg1999] Object Management Group

"OMG Unified Modelling Language Specification", Rational Software Corporation website, June 1999, <http://www.rational.com/media/uml/post.pdf>

[rni1992] "British Braille"
Royal National Institute for the Blind, Peterborough 1992.

[rni2000a] Royal National Institute for the Blind
"Office of National Statistics population estimates" (title truncated), RNIB website, December 2000, <http://www.rnib.org.uk/wesupply/fctsheet/authuk.htm>

[rni2000b] Royal National Institute for the Blind
"Causes and prevention of impaired vision", RNIB website, December 2000, <http://www.rnib.org.uk/info/causes.htm>

[rni2001] Royal National Institute for the Blind
"About Braille and Moon", RNIB website, August 2001, <http://www.rnib.org.uk/wesupply/fctsheet/aboutbm.htm >

[sch1999] Schroeder, Frederic K
"Braille Usage: Perspectives of Legally Blind Adults and Policy Implications for School Administrators", Chapter 2, National Federation of the Blind website, November 1999,
<http://www.nfb.org/braille/chap2.htm>

[seg2001] Segan, Sascha
"Microsoft to Appeal Breakup Ruling in Court", ABC News website, February 2001, http://abcnews.go.com/sections/scitech/DailyNews/microsoft_010223.html

[sha2001] Shankland, Stephen
"Linux growth underscores threat to Microsoft", CNET websites, February 2001, <http://news.cnet.com/news/0-1003-200-4979275.html>

[shi2000] Shirazi, Jack
"'Any Java program can run fast' says author", O'Reilly Publishing website, October 2000, <http://press.oreilly.com/javapt.html>

[shi2001] Shirazi, Jack
"Optimising Hash Functions for a Perfect Map", O'Reilly Publishing website, January 2001, < http://www.onjava.com/pub/a/onjava/2001/01/25/hash_functions.html>

[sla1990] Slaby, W. A
"Computerized Braille translation", *Journal of Microcomputer Applications*, Vol 13, pp 107 – 113, 1990.

[sos1999] Sosnoski, Dennis M
"Java performance programming", Javaworld website, November 1999, <http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance_p.html>

[sul1982] Sullivan, E

"Braille Translation" in *Uses of Computers in aiding the Disabled*, pp 351 – 336 edited by Raviv, J. Amsterdam, 1982.

[sul1997] Sullivan, Joseph E

"A perspective on Braille Unification", National Federation of the Blind website, August 1997, http://www.nfb.org/bm/bm97/unify.htm

[sun1997] Sun Microsystems

"Java Native Interface Specification", Sun Microsystems website, May 1997, <http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html>

[sun1999a] Sun Microsystems

"What is the Java Platform?", Sun Microsystems website, October 1999, <http://java.sun.com/nav/whatis/?frontpage-shortcuts>

[sun1999b] Sun Microsystems

"The Java Hotspot Performance Engine Architecture", Sun Microsystems website, October 1999, <http://java.sun.com/products/hotspot/whitepaper.html>

[sun2000] Sun Microsystems

"Connected, Limited Device Configuration Specification 1.0", Sun Microsystems website, May 2000, <http://java.sun.com/aboutJava/communityprocess/first/jsr030/>

[sun2001a] Sun Microsystems

"Java 2 Platform, Micro Edition (J2ME)", Sun Microsystems website, April 2001, <http://java.sun.com/j2me/>

[sun2001b] Sun Microsystems

"Java 2 Platform API Class Hashtable", Sun Microsystems website, September 2001, http://java.sun.com/j2se/1.4/docs/api/java/util/Hashtable.html

[sun2001c] Sun Microsystems

"EmbeddedJava Technology", Sun Microsystems website, September 2001, http://www.sun.com/software/embeddedjava/

[sun2001d] Sun Microsystems

"PersonalJava Technology", Sun Microsystems website, September 2001, <http://www.sun.com/software/communitysource/personaljava/>

[sun2001e] Sun Microsystems

"Java 2 Runtime Environment, Standard Edition", Sun Microsystems website, September 2001, <http://java.sun.com/j2se/1.3/jre/download-windows.html>

[sun2001f] Sun Microsystems

"StarOffice 5.2", Sun Microsystems website, September 2001, <http://www.sun.com/staroffice/>

[sun2001g] Sun Microsystems

“Enterprise JavaBeans”, Sun Microsystems website, September 2001, <http://java.sun.com/products/ejb/>

[sun2001h] Sun Microsystems
“Java Remote Method Invocation”, Sun Microsystems website, September 2001, <http://java.sun.com/products/jdk/rmi/index.html>

[uni2001a] Unicode, Incorporated.
"What is Unicode?", Unicode Incorporated website, September 2001, <http://www.unicode.org/unicode/faq/utf_bom.html>

[uni2001b] Unicode, Incorporated.
"UTF & BOM", *Unicode FAQ*, Unicode Incorporated website, March 2001, <http://www.unicode.org/unicode/faq/utf_bom.html>

[wal2001] Wall, Larry
“Perl”, Perl Man page, Perldoc website, visited September 2001, <http://www.perldoc.com/perl5.6/pod/perl.html>

[wer1982] Werner, H
“Automatic Braille production by means of computer”, in *Uses of Computers in Aiding the Disabled*, edited by Raviv, J, pp 321-326, Amsterdam, 1982.

[wes2001] Westling, Bjorn; Eriksson, Yvonne; Fellenius, Kerstin & Stigell, Anne.
“Braille and Languages”, Swedish Braille Authority website, August 2001, <http://www.tpb.se/english/Braille_Authority/language.htm>

[wil2001] Wilson, Steve & Kesselman, Jeff
“Java Platform Performance. Strategies and tactics”, Sun Microsystems website, 2001, < http://java.sun.com/docs/books/performance/1st_edition/html/JPTitle.fm.html>

[win2001] WinBraille
April 2001, <http://www.indexBraille.com>

[www1998] World-Wide-Web Consortium (W3C)
“Document Object Model (DOM) Level 1 Specification”, W3C website, October 1998, <http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/>

[zeh1999] Zehner, Ed
<EdZehner@aol.com > “Chinese Braille”, forwarding email from Marco Cimarosti, <Marco.Cimarosti@icl.com> Online posting, 29 October 1999, National Federation of the Blind website, <http://www.nfbnet.org/weblist/blindkid/msg01003.html>