THE COMPUTATIONAL COMPLEXITY OF WRITING SYSTEMS

Gary F. Simons
Summer Institute of Linguistics

Ours is a multilingual world, and the promise of universal computing cannot be realized until we have an adequate solution to the problem of multilingualism in the computing environment. Most present day efforts by the industry to address this problem could be better termed efforts to internationalize computing. Their aim is to produce versions of computers, operating systems, and key programs which work in other languages. The results are single-language products which are just as limited as the original English-based versions. But in the same way that people are multilingual, our computing systems need to be multilingual. We need computers, operating systems, and programs that can potentially work in any language and can simultaneously work with many languages at the same time.

The most visible feature of a language is its system of writing, and the inability to support this for a new language is the first multilingual problem the prospective computer user is likely to encounter. In the typical mainframe or personal computer there is no choice; the graphic encoding of character shapes is an immutable feature of the hardware. The situation is improved somewhat in present day graphics adaptors for personal computers which allow the user to download the definitions of characters shapes. In an operating system like MS-DOS, however, one is still stuck with a one-to-one mapping from character codes to graphic shapes and thus has a limit of 256 possible characters.

The newer technology of bit-mapped graphics has the potential of changing all this, and this kind of technology is assumed for the remarks in this paper. In a bit-mapped system, every dot on the screen is independently controlled by software. This makes it possible to draw any conceivable character in any size or style. The Apple Macintosh is the consumer-level personal computer which has furthest developed the multilingual potentials of this technology. Built into the operating system is the Font Manager (Apple 1985) which makes it possible to have many fonts resident at the same time, each of which can redefine the shapes of the displayed characters. But writing systems involve more than just one-to-one mappings from characters to shapes. Context-sensitive mappings which result in phenomena like positional variants and ligatures are common. Therefore, in 1987 Apple introduced the Script Manager (Davis 1987, Apple 1988) which was designed to handle such aspects of writing systems.

The Script Manager, while far and away the best that is available today, falls short in two respects. (1) It does not provide a general mechanism for defining new scripts. Ideally, a new script could be defined by a user. Instead, each new script must be developed as a special case by a system programmer. In fact, Apple has

not published documentation on how to implement a script system because they have not yet generalized the internal interface between scripts and the operating system. (2) While the Script Manager has a reasonably adequate model of how to display different scripts, it has a very inadequate model of multilingual computing in general. In particular, it lumps into the single notion of script, what should be the three independent notions of language, writing system, and script.

The bottom line is that personal computer users still have no access to a multilingual computing environment which is either truly general or conceptually adequate. Why has multilingualism proven such a difficult problem for the computing industry? Does the wide variety in writing systems of the world possess such inherent computational complexity as to render a general approach impossible?

In the spirit of Geoffrey Sampson's recent book called *Writing Systems*, in which he calls for linguists to take seriously the study of writing systems as a legitimate part of our discipline (1985:11-15), I am setting out to apply a linguistic perspective to the search for a solution to the problem of multilingual computing. In part 1, I establish a baseline by illustrating the range of phenomena which an adequate solution must account for. Then in part 2, I propose a general solution to the complexities of rendering writing systems. Finally, in part 3, I begin to build a conceptual model for the support of multilingualism in the computing environment.

1. Establishing the baseline

To establish a baseline for this study, it was necessary to compile a list of the complexities that occur in writing systems. For a sample, I used Nakanishi's *Writing Systems of the World* (1980). This book identifies and describes 29 major scripts in common use today. His criterion for including a script was that it is used for publishing a daily newspaper. In his definition, a script may be shared by many languages. For instance, the Roman alphabet counts as just one script, which happens to be used in many modified forms by hundreds of different languages.

While there is tremendous diversity in the graphic symbols used in the 29 scripts, there is not very much conceptual diversity. This is not really surprising since all of the 29 scripts descend from one of only two traditions: the logographic system of ancient China (3 scripts) or the alphabetic system of the ancient Semites (26 scripts). All scripts share the same basic logic, namely, that graphic signs which correspond to linguistic signs are written in a sequence. In general, the written sequence is strictly linear, the order of the written signs matches the spoken order of the corresponding linguistic signs, and the graphic signs have a basic form which does not vary.

The complexities in writing systems occur when these generalizations are violated, namely, when the written sequence is nonlinear or out of spoken order, or when graphic signs have variant forms. Figure 1 illustrates the kinds of such complexities found in the sample of 29 major scripts. Before discussing the figure, we need a term to denote the elementary units of a script. I am following Sampson's terminology (1985:22) in calling them *graphs*.

Figure 1 Complexities of writing systems

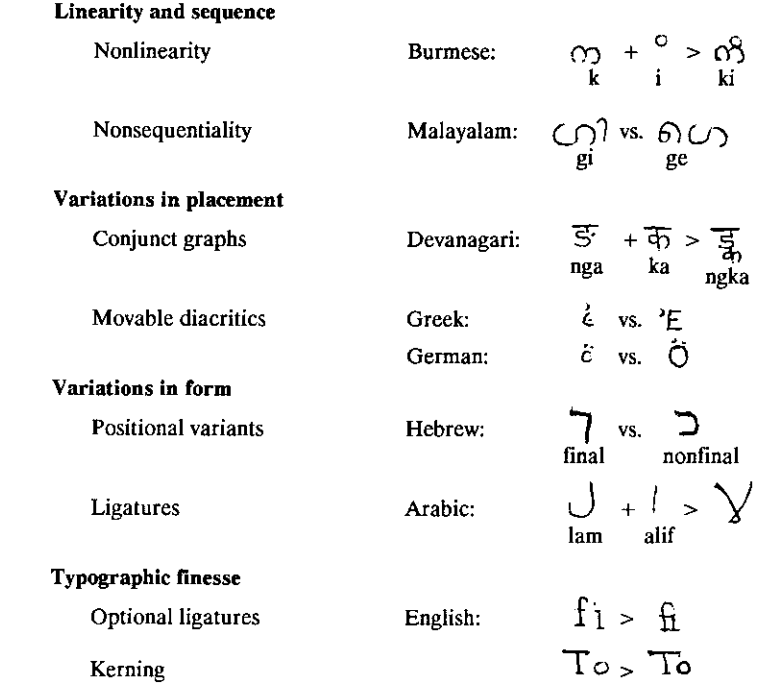


Figure 1 first illustrates cases in which the linearity and sequencing of graphs is skewed from the phonetic. The Burmese example shows a case in which the vowel graph is not part of the linear flow of graphs, but is placed above the graph of consonant which it phonetically follows. The *gi* syllable in the Malayalam example shows the vowel graph following the graph for the consonant it phonetically follows. However, for *ge* the vowel graph precedes the consonant graph. Nonlinearity may also be a result of contextually conditioned variations in placement. In the Devanagari example, a consonant graph which would normally be placed in the linear sequence of graphs is slightly modified and placed beneath another consonant graph when the two sounds combine to form a cluster. The positioning of diacritics is typically affected by context. For instance, the rough breathing in Greek goes over lower case graphs but precedes upper case ones. The umlaut in German must be raised to fit over an upper case vowel.

In many writing systems, a given letter of the alphabet is realized by different graphs in different contexts. For instance, figure 1 illustrates the word-final and nonfinal graphs for the Hebrew letter *kap.* Another kind of variant is the ligature in which adjacent graphs merge into one another to become a distinct graph which simultaneously realizes two elements. An example is given from Arabic. Optional ligatures are also used in high quality typesetting to add a touch of finesse to the

printed copy. For instance, in some high quality fonts for English, the combination *fi* is realized by a ligature which merges the letters together to form unique graphic images which must be defined as a distinct graph in the typesetters font. Another example of finesse in typesetting is kerning. Certain pairs of adjacent graphs leave noticeable gaps of white space in the flow of text. The adjacent graphs may be kerned, or moved more closely together, in order to avoid this. For instance, in a sequence like *To*, the *o* is moved closer to the *T* so that the *T* actually overlaps it.

2. A general solution to the complexities of character rendering

I now propose a general solution for the graphic rendering of all the above writing system complexities. One caveat is necessary. This solution is being proposed as a general one only for writing systems that can be rendered typographically. The computational rendering of calligraphic scripts adds further complexities.

2.1 The need to distinguish character from graph

Writing systems demonstrate the same kind of function versus form distinction that is already familiar to linguists from the study of phonology and grammar. When a single letter of the alphabet is written with different shapes in different contexts, it is a case of a single functional element being realized by a number of environmentally-conditioned formal variants. Ligatures in writing systems are an example of the more general phenomena which linguists call portmanteau, that is, two functional elements being realized simultaneously by a single formal element.

While linguists have long been in the habit of basing notations on the functional level rather than the formal, the computing industry has not. The basic unit of textual encoding on computers is conventionally called the *character*. Because most computer display terminals have a built-in one-to-one mapping from character codes to displayed forms, computer users have been forced to encode information at the level of form. That is, to get the correct graphic forms in the correct contexts it has been necessary to store distinct character codes for the distinctive forms.

It was Joseph Becker, in his seminal article "Multilingual Word Processing" (1984), who pointed out the necessity to distinguish function and form in the computer implementation of writing systems. He observed that character encoding should consistently represent the same information unit as the same character. He then defined *rendering* as the process of converting the encoded information into the correct graphic form for display. He observed correctly that for any writing system, this conversion from functional elements to formal elements is defined by regular rules, and therefore the computer should perform this conversion automatically.

The proper modeling of writing systems thus requires a two-level system. At the functional level are information units called characters. At the formal level are elements called graphs. The higher level units called characters are realized by the lower level units called graphs.

2.2 Mapping from characters to graphs with finite state machines

Becker (1984) stopped short of offering a general computational model for the rendering function. Though at first blush the vagaries of positional variants, nonsequentiality, conjuncts, and ligatures seem to pose a complex problem for multilingual computing, it turns out that there is a simple and general way to implement these phenomena. It uses one of the most fundamental models in computer science, namely, the finite state machine. Finite state machines (also known as automata) have been known to linguists since Chomsky dismissed them as an inadequate formal model for natural language syntax in *Syntactic Structures* (1957). Though they are clearly not powerful enough for natural language syntax, they are foundational in formal language theory and mathematical linguistics. They are presented for linguists in works such as Gross 1972 and Partee 1978.

My inspiration for seeing finite state machines as the general solution to the problem of rendering in writing systems comes from recent work in computational linguistics. Johnson (1972) first observed that the rules of generative phonology could be modeled computationally by a finite state machine called a finite state transducer. A transducer reads an input tape and follows the grammar embodied in its internal states and transitions to write an output tape which is a translation of the input. Kay (1983) developed this idea further, and Koskenniemi (1983) was the first to implement it in a generalized morphological analyzer (see also Karttunen 1983, Karlsson and Koskenniemi 1985). His analyzer has a two-level phonological component which maps from lexical phonemes to surface phonemes (and vice versa) by means of rules coded as finite state transducers. Hegazy and Gourlay (1986) have already observed that finite state techniques can be used in handling the context-sensitive features of the Arabic writing system.

Figures 2 through 5 give an example for a small subset of the Greek writing system. Figure 2 defines the characters, graphs, and mappings for this miniature writing system. The writing system involves four characters, three of which are letters and the last of which represents word breaks. This miniature writing system renders the characters with six graphs: four of them are the default renderings of the four characters, and two of them are variant renderings required in certain contexts. In particular, when *sigma* precedes *space* it has a special final form, and

Figure 2 A small subset of the Greek writing system

```
Characters:

   iota, sigma, omega, space

Graphs:

        ι , σ , ω , ␣ , ψ , ς

Mappings:
      iota  --> ι
      sigma --> σ
      omega --> ω
      space --> ␣

except,
      sigma space --> ς ␣
      omega iota  --> ψ
```

when *iota* follows *omega* the pair are rendered with a single graph which puts an *iota* subscript under the *omega*.

The six replacement rules in the "Mappings" part of figure 2 describe the relationship between characters and graphs completely and unambiguously, provided we require that the longest left-hand parts always match first. Thus, for instance, *sigma space* must always be recognized as the two-character sequence matched by the fifth rule, and never as the single character matched by the second rule followed by the character matched by the fourth character. This means that when we see a *sigma* we can't produce any output until we move to the next character to see if it is a *space* or not; it is not until we see the next character that we will know whether to match the *sigma* with rule two or with rule five.

A finite state transducer is a machine which can embody this kind of matching logic as it reads an input stream of characters to produce the corresponding output stream of graphs. The rules of figure 2 translate into the finite state transducer given in figure 3. The circles in the diagram are called *states*; the arcs are called *transitions*. The machine always begins matching in state 1. Out of state 1 emanate four transitions, one for each of the possible characters in our miniature writing system. The machine looks at the next character in the input stream and follows the transition which is labeled by that character. In figure 3, if the next character is *iota* or *space* the machine returns to state 1. If it is a *sigma*, the machine transitions to state 2, and so on. Below each arc there is optionally an output given in angle brackets. The output consists of one or more graphs of the writing system. When the machine makes a transition, the graphs associated with that transition are output. The diamond-shaped boxes in the figure are references to existing states of the machine. Rather than drawing a number of arcs that would loop back to the left and clutter up the diagram, the diamonds are used instead to say, "Go back to the designated state."

Figure 3 A finite state transducer for the Greek subset

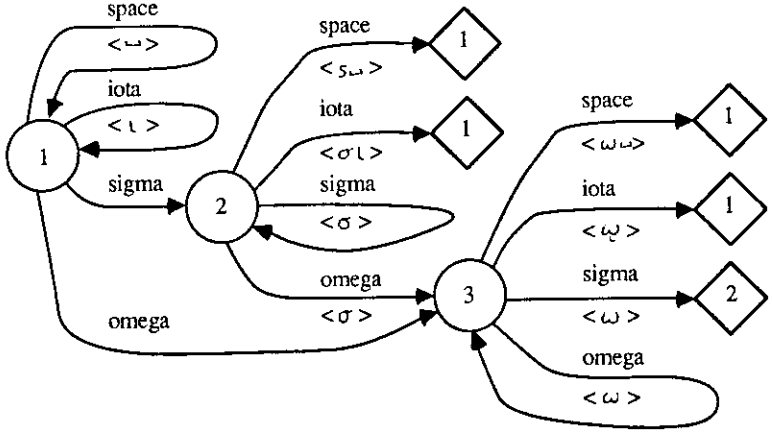


Figure 4 gives an example of the transducer in use. The input string *iota sigma omega iota sigma space* is transduced to generate the graphic sequence: ιϲωϛ␣. The "State" line begins with 1 to show that the machine starts in state 1. From then on it indicates the state into which the machine moves after recognizing the character immediately above it. "Output graphs" shows the graphs which are output when the transition to the state above is made.

Figure 4 A sample output from the finite state transducer

Input characters:		iota	sigma	omega	iota	sigma	space
State:	1	1	2	3	1	2	1
Output graphs:		ι	ϲ	ω			ϛ␣

To complete this example, the same finite state transducer is given in figure 5 as a transition table. A table like this is often used to represent a finite state machine in a computer implementation. The row labels are all the states in the machine. The column labels are all the possible input characters in the writing system. The body of the table tells for each possible combination of current state and next input character, what the next state is and what graphs should be output if any. Comparison of figures 3 and 5 will show that the table contains the very same information as the state diagram.

Figure 5 The finite state transducer as a transition table

	iota	sigma	omega	space
1	1,<ι>	2	3	1,<␣>
2	1,<ϲι>	2,<ϲ>	3,<ϲ>	1,<ϛ␣>
3	1,<ω>	2,<ω>	3,<ω>	1,<ω␣>

This example, though quite simplified, illustrates how a finite state transducer can handle the phenomena of positional variants and ligatures. Nonsequential rendering is handled in the very same way by implementing mapping rules of the form, *ab --> ba*. Thus we see that all these difficulties for text rendering have been reduced to a single generalized solution. From the standpoint of computational complexity this result is particularly significant because the finite state machine is one of the most efficient known in computer science. It is guaranteed to produce its output in a time directly proportional to the number of characters in the input string. It is called a linear-time problem, because a graph of execution time versus length of input string would be a straight line. In terms of computational complexity, this algorithm for implementing the many-to-many mappings of textual rendering is just as efficient as the unacceptable one-to-one mapping scheme built into most current computer systems. The price we pay for the added capability is the extra memory space in which the finite state machine must be stored.

Building the finite state machine for a full writing system could be a tiresome and error-prone task. Fortunately, computer science has a solution. Algorithms for converting rules systems (such as given in figure 2) into finite state machines are well known. Thus it is my proposal that a generalized implementation of writing systems would allow users to describe new writing systems by expressing the mapping from characters to graphs as rewrite rules. The computer could take over from there to compile that description into a very efficient implementation as a finite state transducer.

2.3 The need to distinguish graph from image

Thus far we have spoken of graphs as the formal realization of characters. But in fact, graphs are not the ultimate concrete realization; they, too, have variants. For instance, there are thousands of written or printed symbols which a user of a Latin-based script would recognize as the graph called "lower case *a*." These variants could differ in size (such as 12 points versus 18 points), weight (such as medium versus bold), posture (such as regular versus italic), aspect (such as normal versus condensed), or in style (such as Times or Helvetica).

I shall use the term *image* to denote the concrete graphic form that a graph ultimately takes on the computer screen or on paper. Each of the above mentioned dimensions in which an image can vary is called an *attribute*. The combination of attributes which describe a particular image of a graph, I will refer to as the *appearance* of the graph. The complete set of images for a given appearance is called a *font*.

A font is another instance of a mapping from higher level functional elements to lower level formal ones. And like the mapping from characters to graphs discussed above, the mapping from graphs to images must allow for context-sensitive variants. This is in order to implement the phenomena I referred to as "typographic finesse" in section 1, namely, font-specific ligatures and kerning. These do not belong in the higher level mapping from characters to graphs because because they are not used in all fonts; they are font-specific details.

Kerning and font-specific ligatures can be generalized under a single mechanism, namely, the mapping mechanism we have already discussed for the translation of characters to graphs. Each font may specify a private mapping function which defines the kerns and ligatures specific to the font. In the case of kerns, this means that the replacement string in mapping rules must include positioning commands as well as graph codes.

The addition of positioning commands allows us to solve the only remaining rendering problem, namely, the placement of diacritics. This is handled by mapping rules which translate combinations of diacritic graphs with a particular base graph into the proper images plus horizontal and vertical positioning commands to put the diacritics in exactly the right place with respect to the base character image and to each other. This kind of positioning belongs in the font-specific mapping because different appearances for the same base-diacritic pair could require different positioning commands.

2.4 Mapping through multiple levels with a single finite state machine

Our model now has three levels. The writing system deals with the relationship between the top two; it maps from the character codes of a language to graph codes of a script. The font system deals with the relationship between the bottom two; it maps from the graph codes to displayed graphic images. Both mappings can be implemented by the same general approach of finite state transduction.

By adding another level to the mapping operation, it appears that we are adding to the computational complexity of the task. In fact, this need not be the case. One of the results of automata theory in computer science is that a sequence of finite state transducers which map an input string through a series of levels, can always be converted into a single, larger finite state transducer which will produce exactly the same output while performing only a single transduction on two levels. In making this conversion, the combined finite state machine will take up more space than the total space of all the original ones. In return we get a single finite state machine which optimizes the speed—it is equally as fast as any of the original machines by themselves.

3. Toward a conceptual model for multilingual computing

In the introduction, I stated that the best multilingual system currently available for personal computers, namely the Macintosh Script Manager, offers neither a general way to define new scripts nor an adequate conceptual model of multilingual computing. The preceding section proposes a solution for the first of these; this section addresses the second.

3.1 The need for language-specific character sets

Communication between computer and user is not the only issue in multilingual computing. We must also be concerned with communication of encoded information between users (and their computers). Clearly, there must be agreed upon standards for what each character code represents before users can enjoy unlimited interchange of multilingual information.

One proposal that has come out of linguistic circles (Anderson 1984) is that we develop a universal character set that would include all the characters needed for encoding all languages. The proposal calls for codes which would use 16 bits of computer storage. This would make possible the definition of more than 65,000 characters. This approach, even if it were possible to achieve agreement on the members of the universal character set, suffers from a very basic flaw. It views multilingual information as a sequence of multilingual characters. The most fundamental organizing principle in multilingual information, namely, the notion of language itself, is missing.

Every piece of textual information stored in a computer is expressed in some language. It may in turn contain parts expressed in another language, and so on (such as when an English essay quotes a German paragraph which discusses some Greek words). The language that a particular piece of information is in governs

much more than just the writing system by which it gets displayed. It also governs how new information is typed, how two strings of text are compared to determine alphabetical ordering, how a string of text is decomposed into useful units (like sentences, words, or syllables), and how text is transliterated into other scripts or into other character coding standards.

Thus I propose that the correct way to handle multilingual information encoding is to use a different character set for each language. The key to keeping the languages separate is that the stream of encoded data must contain codes which identify the languages. Fortunately, unlike the universal character set proposal which has yet to be implemented, a multiple character sets approach has been adopted by the International Standards Organization (ISO 1973) and numerous conforming character sets have been established by national standards bodies.

3.2 The need to distinguish language from script

Current multilingual systems fall short of defining the notion of language as one of the key elements of a multilingual information processing system. With their focus on word processing they stop with scripts. I now argue for the need to distinguish language from script.

Scripts are shared by many languages. This is the definition we adopted in section 1 following Nakanishi's usage. This also accords with the use in current multilingual word processing systems where the additional graphs needed for various European languages are not built into new scripts, but are added to the same Latin-based script that is used for processing English.

With this definition of a script as including all the graphs required to render all the languages based on that script, it is possible to equate language and script when it comes to rendering. However, for other aspects of information processing this is unacceptable. For instance, the collating sequence for sorting is language specific; so are the rules for finding syllable breaks. If the computer is to know how to process a given piece of information, that information must be stored with a tag identifying its language. Then the system can consult a definition of how the basic information processing tasks are performed for that language. Script enters in as part of the definition of the rendering task.

Another reason for needing a distinction between language and script is to render text in languages which are written with more than one script. A classic example of this is Serbo-Croatian; the Serbs write their language with a Cyrillic script, while the Croats write the same language with a Roman script. The coding of text in languages like this should be independent of the script which will be used to render it. A user should be able to see the same text rendered in either script without having to change the stored data.

It is clear then that there is not a one-to-one correspondence between language and script. Many languages can be written with the same script. On the other hand, a single language can be written with many scripts. Language versus script must therefore be another two-level distinction in the model of general multilingual computing.

3.3 The need to distinguish writing system from script

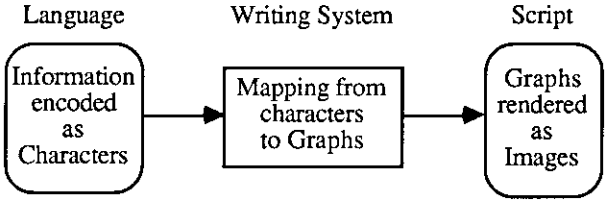
That we should need to distinguish between language and script is not surprising since they are clearly distinct concepts in everyday English-language usage. This is not true of the terms "writing system" and "script." In fact, Sampson (1985:20) states that "writing system" and "script" are used virtually interchangeably in his book. He goes on to discuss the problem of deciding whether two systems of writing exemplify two different scripts or are variants of the same script, posing writing in English versus German as an example (1985:21). In Sampson's treatment this remains an unanswered question.

The formal computational modeling of writing demands an answer, however. In the sense in which German and English writing are the same, the computational model must make them the same. In the sense in which German and English writing are different, the computational model must make them different. We can achieve just this by introducing a distinction between the notions of writing system and script. German and English clearly have different systems of writing, thus we will say that they have different writing systems. However, they are both based on the Latin script; thus we will say that they use the same script. We now develop a more formal definition of writing system versus script.

In this computational model of writing, language is the domain of character encoding. Scripts on the other hand, are the domain of graph rendering. Scripts know nothing about characters, they simply know how to render graphs. For instance, the graph which represents the letter *c* in English is rendered the same way in every language which uses it, regardless of the language-specific character code with which that graph may be associated. The model does not yet have a way of getting from the characters of a language to the graphs of a script. This is where the notion of writing system comes in.

Formally, a writing system is a mapping from the characters of a language to the graphs of a script, as illustrated in figure 6. The fact that a single language can be written with more than one script indicates that the mapping from characters to graphs is not just a property of the language. Rather, that mapping is a conceptual entity in its own right and a language can be associated with more than one of them.

Figure 6 The distinction between writing system and script



The definition of multiple writing systems for a single language solves not only the problem of languages that can be written in more than one script, but also the problem of contemporary and historical variants which are based on the same script. For instance, in writing German it is possible to substitute digraphs with *e* for umlauted vowels. In English 200 years ago, *s* was written with final and nonfinal variants. It is clearly desirable that text in such cases be encoded independently of these variations, so that selecting a particular writing system causes the text rendering to immediately change to the selected style.

Once writing system is defined as a distinct entity, we find two other aspects of multilingual text processing which belong in the writing system. These are the comparison function for sorting and the definition of directionality. The comparison function cannot be a property of the script, because different languages which use the same script have different sorting conventions. Nor can the comparison function be a property of the language. When a single language is written in multiple writing systems, there are generally different sorting conventions associated with the differences in writing system. Thus, the sorting component must be a property of the writing system.

Similarly, the direction of writing must be a property of the writing system. It is not a property of the language, for the same language written in two different scripts like Roman and Arabic will switch from left-to-right or right-to-left depending on the script. However, direction is not strictly a property of the script either. For instance, the Korean language written in the Hangul script is printed either in vertical top-to-bottom lines or in horizontal left-to-right lines (Nakanishi 1980:93). This fact would be modeled by defining two different writing systems. The direction component must also be able to specify a boustrophedon style of writing (in which the direction of successive lines alternates), and the mirroring or rotation of graph images which happens when line direction is reversed or when normally horizontal scripts are written vertically.

3.4 The need to view script as font family

In section 2.3 above we defined a font as a mapping from graphs to images. We have been defining a script in the same way. The basic difference is that a font can render graphs in only one appearance. A script can conceivably render graphs in thousands of appearances. Essentially the job of a script is to determine which font renders the requested appearance and to direct the rendering request to that font.

To organize the potential complexity of a script involving thousands of fonts, the fonts should be organized into a hierarchy of font families. A tremendous amount of duplication of information in closely related fonts can be avoided by introducing the convention that fonts (and all the fonts in a font family) inherit the characteristics of their parent font family. Thus shared characteristics need be defined only once for the font family rather than being repeated in every affected font. Whatever can be specified about a font can be specified about a font family—graphic images, metric information about the sizes of rendered images, new graphs for font-specific ligatures, mappings for ligatures or kerns. In the case of the graphic images, image definitions for a specific font could be instructions to

inherit the images from the parent font family and then scale them or otherwise transform them in a systematic way.

If a font family can specify all the same information as a font, what is the difference between them? Really, there is only one—unlike a font family, a font has no members.

And what is a script? In this model, a script turns out to be the same thing as a font family, but with one special characteristic—it has no parent font family. A script is thus the root node from which all the fonts used to render texts in a particular writing tradition inherit their commonality. The essential aspect of that inheritance is the set of graphs which every font of the script can render. As one descends the hierarchy of font families to levels of greater delicacy, the inventory of graphs may grow as greater finesse is applied in the context-sensitive rendering of graphs, but those additional graphs are invisible to the script itself.

We have now disposed of the functionality of the Macintosh-style script altogether. The font-specific ligatures belong to the private mapping functions of fonts or font families. The character-to-graph mappings that embody the obligatory ligatures, context-sensitive variants, and nonsequential ordering belong to the writing system component. Text processing functions like typing and string decomposition belong at the language level. The comparison function and directionality belong in the writing system.

3.5 Putting it all together

The conceptual model for multilingual text processing developed here is summarized in figure 7. It makes use of a semiformalized conceptual modeling language to do so. The definitions of the elements in the model have three basic parts. The first part is a header line which uses boldface type to name the element being defined. The second part is the list of components of the element. The third part, given in parentheses, defines what type of element each of the components must be. The names of elements which are defined elsewhere in figure 7 are given in italics. (The element type *string* refers to the normal sense in computing of an arbitrary sequence of characters.)

Figure 7 A conceptual model of languages and writing systems

A **language** has
- a name (which is a string),
- a code (which is a number),
- a character set (which is a set of *characters*),
- a typing component (which is one or more typing systems),
- a rendering component (which is one or more *writing systems*),
- a decomposition component (which is a set of decomposition functions), and
- a transliteration component (which is zero or more transliteration systems).

A **character** has
- a name (which is a string), and
- a code (which is a number).

A **writing system** has
a name (which is a string),
a code (which is a number),
a script (which is a *script*),
a mapping (which is a set of *character mapping rules*),
a direction component (which specifies direction of graphs in line, or lines on page, and mirroring or rotation of graphs), and
a sorting component (which is a comparison function).

A **character mapping rule** has
a "from" part (which is a sequence of one or more character codes), and
a "to" part (which is a sequence of zero or more graph codes).

A **script** is a *font family* which has no parent.

A **font** is a *font family* which has no members.

A **font family** has
a name (which is a string),
an appearance (which is an *appearance specification*),
a parent (which is a *font family* or empty),
members (which is a set of *fonts* or *font families* or empty),
new graphs (which is a set of *graphs*),
a mapping (which is a set of *graph mapping rules*),
graphic definitions (which define basic graphic images for all graph codes),
height definitions (which define the rendered height for the images), and
width definitions (which define the rendered width for the images).

An **appearance pattern** has
a style (which is a list of possible styles),
a posture (which is a list of possible postures),
a weight (which is a list of possible weights),
an aspect (which is a list of possible aspects),
a size (which is a list of possible sizes).

A **graph** has
a name (which is a string), and
a code (which is a number).

A **graph mapping rule** has
a "from" part (which is a sequence of one or more graph codes), and
a "to" part (which is a sequence of one or more graph codes or positioning commands).

With the full conceptual model in place it is now possible to define the rendering function of a multilingual computing environment. Informally, the function would be something like this. To render a string in a particular appearance, the system finds the language and writing-system tags of the string, and then looks up the corresponding writing system definition in the system resources. The mapping function defined in the writing system is used to convert the string of character codes into a string of graph codes. That string of graph codes is then submitted to the script specified in the writing system definition in order to

generate the graphic images. The font hierarchy of the script is searched to find the definition of the font which renders the desired appearance. As the font hierarchy is descended to more delicate levels, the graph sequence is submitted to the font-specific mappings until the bottom level is reached and the proper graphic images are drawn in the correct positions.

This traversal of the font hierarchy and cascading of mapping operations sounds like it could be computationally complex and inefficient. But as discussed in section 2.4, it need not be. A compiler could translate a multilevel description of a writing system and its script into a single finite state transducer which performs the mapping from characters to images in linear time. In terms of computational complexity, this puts the problem at the very bottom of the scale.

My conclusion, therefore, is that the chief obstacle to the generalized implementation of writing systems in a multilingual computing environment has not been inherent computational complexity. Indeed, we have shown that the problem can be reduced to a very simple one in that regard. Rather, the chief obstacle has been the conceptual complexity of the problem. In particular there has been a failure to recognize language and writing system as two essential elements that are distinct from each other and are distinct from script. I am hopeful that this linguistic perspective on the conceptual basis of multilingual computing will help lead us to the solution of a problem which continues to plague our use of computers in a multilingual world.

References

Anderson, Lloyd B. 1984. Multilingual text processing in a two-byte code. *Proceedings of Coling84, Stanford University, 2-6 July 1984.* Morristown, NJ: Association for Computational Linguistics. Pages 1-4.

-----. 1985. Multilingual word processing systems: desirable features from a linguist's point of view. *Newsletter for Asian and Middle Eastern Languages on Computer* 1(1):28-30.

Apple Computer. 1985. The Font Manager. In *Inside Macintosh,* volume 1, pages 215-240 (with updates in volume 4, pages 27-48, 1986). Reading, MA: Addison-Wesley.

-----. 1988. The Script Manager. In *Inside Macintosh,* volume 5, pages 293-322. Reading, MA: Addison-Wesley.

Becker, Joseph D. 1984. Multilingual word processing. *Scientific American* 251(1):96-107.

Chomsky, N. 1957. *Syntactic Structures.* The Hague: Mouton.

Davis, Mark Edward. 1987. The Macintosh script system. *Newsletter for Asian and Middle Eastern Languages on Computer* 2(1&2):9-24.

Gross, Maurice. 1972. *Mathematical Models in Linguistics.* Englewood Cliffs, NJ: Prentice-Hall.

Hegazy, W. A. and John J. Gourlay. 1986. Handling context sensitivity in Arabic script. In J. J. H. Miller (ed.), *PROTEXT III: proceedings of the Third International Conference on Text Processing Systems.* Dublin: Poole Press. Pages 125-130.

ISO. 1973. *7-bit Coded Character Set for Information Processing Interchange* (ISO 646-1973), and *Code Extension Techniques for Use with the ISO 7-bit Coded Character Set* (ISO 2202-1973). Geneva: International Standards Organization.

Johnson, C. Douglas. 1972. *Formal Aspects of Phonological Description.* Monographs on linguistic analysis, number 3. The Hague: Mouton.

Karlsson, Fred and Kimmo Koskenniemi. 1985. A process model of morphology and lexicon. *Folia Linguistica* 14:207-231.

Karttunen, Lauri. 1983. KIMMO: a general morphological processor. *Texas Linguistic Forum* 22:163-186.

Kay, Martin. 1983. When meta-rules are not meta-rules. In Karen Sparck Jones and Yorick Wilks (eds.), *Automatic Natural Language Parsing.* Chichester: Ellis Horwood Ltd. Pages 94-116.

Koskenniemi, Kimmo. 1983. *Two-Level Morphology: a general computational model for word-form recognition and production.* Department of General Linguistics, University of Helsinki. Publication number 11.

Nakanishi, Akira. 1980. *Writing systems of the world: alphabets, syllabaries, pictograms.* Rutland, VT: Charles E. Tuttle Co.

Partee, Barbara Hall. 1978. *Fundamentals of mathematics for linguistics.* Stamford, CT: Greylock Publishers.

Sampson, Geoffrey. 1985. *Writing Systems: a linguistic introduction.* Stanford, CA: Stanford University Press.