Universal Access in the Information Society manuscript No.
(will be inserted by the editor)

Transformation frameworks and their relevance in universal design

Silas S. Brown and Peter Robinson

University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
e-mail: {Silas.Brown,Peter.Robinson}@cl.cam.ac.uk

Received: date / Revised version: date

Category: Long Paper

Key words notations, transformation, conversion, education, tools, 4DML

Abstract Music, engineering, mathematics, and many other disciplines have established notations for writing their documents. Adjusting these notations can contribute to universal access by helping to address access difficulties such as disabilities, cultural backgrounds, or restrictive hardware. Tools that support the programming of such transformations can also assist by allowing the creation of new notations on demand, which is an under-explored option in the relief of educational difficulties.

This paper reviews some programming tools that can be used to effect such transformations. It also introduces a tool, called "4DML", that allows the programmer to create a "model" of the desired result, from which the transformation is derived.

1 Introduction

The transformation of data from one structure to another is a recurring theme in computer science and software engineering:

- Compilers and other software development tools are based on transforming the programmer's input into an executable form (or, in the case of generative programming, transforming it into lower-level code that is to be compiled).
- Different programs (and in some cases items of hardware) use different data formats to represent the same thing, so conversion is often needed when exchanging data between them. For example, there are many converters between different document formats, different sound and image file formats, musical score formats, etc. Object request brokers (ORBs) and middleware frequently employ conversion.
- Some algorithms can be simplified if the data is first transformed into a convenient structure. Many algorithms can be regarded as transformations in their own right.
- Transformation can be important when presenting data to the user and when accepting user input.

This last point, namely the importance of transformation in user interaction, is relevant to universal design and will be elaborated here.

1.1 Transformation in universal design

Universal design aims to develop "technologies which are accessible and usable by all citizens... thus avoiding the need for *a posteriori* adaptations or specialised design" [37]. In his closing plenary address at ACM CHI 2001, Vanderheiden pointed out that this does not mean designing one homogeneous user interface to fit everybody, since people have conflicting requirements and a "lowest-common denominator" interface would be limited indeed:

> "It can't have a *visual* interface [because some people can't see]; it can't have an *audio* interface [because some can't hear];... you could design a *brick*..."

Hence, a technology that fulfils (or partly fulfils) the aims of universal design is likely to include transformation functionality. This is not specialised design if it is customisable and extensible, and it is not *a posteriori* adaptation if the transformation needs are foreseen *a priori* in the original design.

1.2 Diversity and the need for general solutions

A popular misconception is that individual needs and requirements are nearly identical, and are adequately addressed by existing systems and hence there is no need of further research. This section aims to show otherwise.

Individual needs and requirements are diverse and can be difficult to anticipate. A person's ability to use data in printed form may be hampered by a print disability, such as blindness, low vision, or dyslexia. It may also be hampered by educational and cultural differences. For example, the person may have learned a notation that is different from the one being used in the presentation. Motor disabilities may limit data entry and interactive navigation with notations.

Diversity of print disabilities. The diversity of print disabilities is frequently glossed over in the literature. Many papers use the phrase "blind and visually impaired" when discussing designs for blind people, implying that blind people and people with low vision have similar interaction requirements; in fact, many with low vision use their residual sight as much as they can [25]. Other papers (e.g. Hermsdorf et al. [16]) assume that the needs of all partially-sighted people are much the same. Jacko et al. [17, 18] show that this is not the case, and suggest that individual users' needs can be determined by clinical assessment. Even this can be difficult in the case of some eye conditions, such as nystagmus, which can vary over time and can produce different perceptual results for different people [39]. Some users need to be given control over the presentation themselves, as Gregor and Newell explain for some cases of dyslexia [14].

Many industry-standard applications already allow the customisation of fonts, sizes and colours, but it can be difficult and is not always reliable. In the first author's personal experience as a person with low vision, the customisation of applications frequently exposes bugs in the display code; these can render the application unusable and are rarely fixed in a timely manner. Further, few of these applications allow the layout of structured data to be changed to compensate for the reduced viewing area to text size ratio (see Section 2.2). Different layout algorithms can sometimes make the data easier to navigate around by people who have difficulty fixing their gaze, and this requires additional customisation.

Disabling circumstances. A document may have to be used in unusual circumstances that are effectively disabling. This includes the use of restrictive hardware, such as small displays on mobile telephones, which present a problem similar to that of large print on normal displays, as the ratio between the viewing area and the text size is reduced, and therefore layout and navigation need careful consideration. Adjusting the notation can help in this case also.

Specialist notations. Educational background can contribute to the requirement for an alternative presentation. A good example of this is in music. Besides Western staff notation, musicians use various tablature and instrument-specific notations, as well as *sol-fa*, Chinese *Jianpu* notation, and others, and it is often possible to transcribe a piece of music from one notation into another in order to make it accessible to a greater number of musicians. Braille music also has numerous different versions across the world.

Notations can be customised for different tasks, such as sequential reading, rapid overview, or detailed analysis. Often it is desirable to omit or include certain details depending on how the document will be used, because people with print disabilities are frequently unable to skip over unwanted information at speed. Additionally, educational establishments can customise notations for pedagogical purposes.

Data entry is another task that can call for alternative notations, since optimising for input and editing is different from optimising for reading. Even direct-manipulation interfaces sometimes use a hidden input notation in their controls. For the sake of usability, there is usually some compromise so that the input and output notations are conceptually similar, but this can be overshadowed by a disabled user's accessibility needs (for example, someone with extreme typing difficulties might prefer a terser input notation even if it means more training).

Multiple and unforeseen needs. Although it is common practice to focus on one need at a time, it is possible to envisage individuals who have a combination of several of the above-mentioned requirements, as well as additional ones that have not been anticipated by research. Hence, the need arises for general solutions, i.e., systems that truly include as many people as possible by being adaptable to new situations that were not explicitly considered at design time.

2 Special-case transformation systems

This section discusses some illustrative examples of systems that use transformations to assist people with disabilities. The systems are "special case", not because their primary purpose is to support disabled users, but because the transformations they employ are limited to one or two domains (such as Web pages, or mathematics) and, although they might permit some customisation, they cannot easily be programmed to handle completely new transformation tasks. This can be the case even in systems that are examples of universal design.

2.1 Assistive technologies for print disabilities

In 1972, Rubinstein and Feldman proposed a Braille terminal for blind computer operators [34]. Computer-driven reading machines for the blind were also conceived at that time [2]. During the 1980s, when MS–DOS was the industry standard, many companies (such as Cobolt Systems, Dolphin Systems, Techno-Vision and

PulseData) marketed "adaptive" or "assistive" technologies for blind DOS users. These adaptations usually involved a combination of speech hardware and screen-reading software. Screen magnifiers for partially-sighted people were also common, and some Braille displays were available, although the latter were more expensive and required knowledge of Braille, which not all blind people had [30].

Pitt and Edwards [30] evaluated a screen reader and found several inadequacies. Unlike the line-mode terminals of the 1970s, most DOS applications updated the screen in a highly non-linear fashion. Simply reading everything as it was written was no longer effective, and algorithms had to be developed to allow blind people to extract the appropriate information from the screen without having to read or listen for too long and without having to remember too much context (since verbal information is transient).

Developing such algorithms often involved making specific allowances for commonly-used applications, usually by writing "profiles" for those applications. For example, if a clock was displayed at a certain screen position, specific code had to be written in order to avoid reading the content of the position in question, as such content is updated every second. This led to a situation where blind people (other than programmers) were effectively restricted to certain pieces of software, and certain versions of them, as a trivial change in the software's appearance could require the writing of a new profile.

GUIs. The rise of the GUI during the 1990s rendered DOS screenreaders obsolete, and happened some time before effective replacements were developed [40]. As well as the problems that DOS screenreaders had to face, there was now the additional complication that programs can display their controls by writing bitmaps, instead of using the standard system calls, and thus be more difficult to intercept (this is an example of the problems raised by signal-based representations discussed in Section 3.1). Moreover, the visual concepts of GUIs do not come naturally to computer novices who have been blind from birth. Research interest in screenreaders included the Mercator project for the X window system [28], and the GUIB project for Windows [42], which tries to simulate in a speech environment the direct manipulation of positioning windows.

Navigation. When the amount of data that has to be displayed is significantly greater than what will fit on the display, users have to "navigate" around the data, and can "get lost" (for example, see Watts-Perotti and Woods' article on the subject [41]). The use of screen magnification, speech, Braille, or any other output method that cannot display as much information simultaneously as the software designer expected, can accentuate this problem.

2.1.1 Application-level transformations Because of the above-mentioned problems with assistive technologies, Raman [32, 33], Zajicek et al. [46], and others adopted the approach of implementing specialised applications to cater for the needs of blind people, rather than trying to adapt industry-standard systems that were not originally designed for the purpose. Raman extended the EMACS editor with a speech interface that is completely different from its visual interface. He also produced a comprehensive system for reading mathematical documents as speech (either interactively or non-interactively). Zajicek's work involves a Web browser that provides navigation aids for the blind, involving information-retrieval techniques on complex Web pages.

2.2 Customising Print

Syntax highlighting. Normally used for programming languages, syntax highlighting involves marking up text with colours (or fonts) to indicate its syntax. This helps people with print disabilities, provided that they can see the highlighting, because, once they are familiar with the language and its idioms, they can "zoom out" and navigate around the colour pattern without having to read the details. It also assists with fixation: if, due to physical or other reasons, it is difficult to concentrate on a fixed point on the page, then it is often easier to return to that point by using low-resolution colour information.

Colour highlighting has been effectively used in normal text [14], as well as in music (by *Sibelius* [11]). It is an example of enhancing a notation by adding extra information in a format orthogonal to the original, so as to aid navigation around the notation.

Layout problems. Large text implies that only a part can be fitted into a given area. If the layout is not flexible, then extensive navigation is required, such as laborious horizontal scrolling, or unwieldy large paper that introduces extra requirements on the reprographics facilities.

Even if the layout is flexible, some page-layout algorithms can fail to produce a readable layout, given the extreme constraints that very-large print presents. Such algorithms have a maximum text-size-to-page-area ratio beyond which they break down. This can be demonstrated by viewing a website with frames and tables on a low-resolution screen at the largest font setting.

For this reason, people who use very large print will often wish to transform tabular layouts into a more flexible form (see, for example, the various website mediation technologies that have been produced [6]).

2.3 Braille Typesetting

Algorithms for typesetting Braille text have been developed since the 1980s. Once the Braille codes have

Fig. 1 A refreshable Braille display attached to a normal keyboard (photograph from `www.tieman.de`)

been produced, they can be sent to an embosser, i.e., a device for producing raised dots on suitable material, sometimes called a "Braille printer", or to a refreshable Braille display, which uses mechanical or piezoelectric techniques to temporarily raise the dots (Figure 1).

Some of the more well-known typesetting products are the Royal National Institute for the Blind's "Braille-It!", the US National Federation of the Blind's NFBTrans, and the commercial TuxTrans system (which is multilingual and can also translate some mathematics). The state of the art is probably the high-resolution (20dpi) Tiger Advantage embosser with its Windows printer driver that allows almost any document (including diagrams) to be embossed with little effort, but most establishments have older embossers.

Contractions. The use of "contractions" (abbreviations) can increase the reading speed in alphabetic scripts. This is not just data compression, since many experienced users of Braille associate contractions with phonetic or even semantic concepts (e.g., using the contraction for "mother" in the word "chemotherapy" is a bad idea). Even high-end Braille typesetting products are sometimes overzealous in their use of contractions, particularly with newly-invented words like "scandisk" (which contains "and", so is sometimes incorrectly abbreviated "sc&isk"). The use of speech synthesis algorithms could help to avoid cross-syllable contraction.

Customisability. Most contraction algorithms are rule-based and can be used for different languages if appropriate rules for each language are provided. A problem with many such systems is that it is very difficult for the user to customise the rules, which may need to be done in a pedagogical setting. Blind children learn the contractions and word abbreviations of Braille in carefully-graded steps, and while they are doing this they need special texts that use some contractions but not others [12]. If these texts are to be generated automatically, then the teacher must be able to customise the list of contractions that can be used, preferably without getting lost in a large database of rules and exceptions in an unfamiliar notation.

East Asian languages. Producing Braille from Chinese and Japanese text presents other challenges. There are Braille codes for analytically representing the characters, but most text is more readable when transcribed from one of the phonetic character sets, so the text must first be "read" into sounds [21]. The latter requires natural-language processing, because there is usually more than one way of reading any given character, depending on the context.

Specialist notations. More general problems are associated with specialist Braille notations such as mathematics, musical scores, chemical bonds, and so on. These notations have many different standards and house styles, and most existing transcription software (such as MFB [24] and Goodfeel [26]) is limited to outputting in very few of them. The problem is further complicated by the fact that the source material is stored in many diverse formats and therefore multiple conversions are often required, sometimes leading to information loss due to the limitations of intermediate formats.

3 Generalised transformation frameworks

This section reviews some transformation systems that are programmable, i.e., they can be used as a basis for implementing new transformation tasks as needed. These systems are not limited to one domain, but deal with generalities, such as symbols and data, that can apply to many domains. A generalised system that is used in the field of software engineering, for example, might equally well be used to accomplish transformation tasks when dealing with mathematical or musical notation.

The notion of transforming data from one structure to another is a very general one. Nearly every program ever written can be thought of as a transformer, interpreting its input (perhaps a sequence of commands from the user) and producing some output (perhaps feedback to the user as the input is being given). Conversely, a given transformation could conceivably be implemented in virtually any programming language.

As is the case with many programming problems, however, the class of transformations that form the main focus of this paper, i.e., the conversion and adaptation of the notations of various educational disciplines, can be achieved more easily with some programming tools and languages than with others. While different programmers have different ideas of what is easy and what

is difficult, it is still possible to achieve some generality by stating that, if a tool or language was developed specifically to support a particular design approach that is well-suited to a certain class of problems, then many people who wish to solve problems in that class are likely to find that tool or language easier to learn and use than a more general programming language, and the associated design approach is more likely to be adopted, hence reducing the amount of work that might otherwise result.

Many transformation tools are actually Turing-powerful and could potentially be used as general-purpose programming languages, but it is still useful to make the distinction, because these tools are *biased* toward a particular class of problems and a particular design approach. If they are used outside that class, the resulting code may not seem so high-level, particularly if there is overt *emulation* of the behaviour of other general-purpose programming systems.

Any programmable system must strike a balance between versatility and simplicity. The limit of versatility is to require the user to write the entire program in low-level code, but that gives little simplicity. Conversely, the limit of simplicity is to forbid any programming (just provide built-in special-case functionality), but that gives little versatility. Between these limits lies a complex non-linear relationship that depends on design decisions. These should make the system powerful enough to serve its purpose, but simple enough to justify its use as a tool.

3.1 Symbolic vs signal-based representation

Representations of information in computer memory can generally be divided into two categories: symbolic and signal-based. Symbolic representations are based on the practice of encoding the *identifications* of symbols, whereas signal-based representations are based on *measurements* of some physical quantity (such as light or sound) which may contain information. Scanned images, for example, are signal-based, whereas program code is largely symbol based.

Some representations are symbolic in one sense and signal-based in another. For example, many desktop publishing applications *identify* which symbols are on the page but *measure* their positions (which the user can indicate with a pointing device). This sometimes causes difficulty when a program needs to act on the relationships between different symbols, since those relationships need to be interpreted from the physical positioning data. Similarly, a MIDI file generated by a music keyboard *identifies* which notes are being played but *measures* their positions in time. If the music is to be written in staff notation then these time measurements must be interpreted.

Signal-based representations may lead to more accurate reproduction of an original source, but they limit the effectiveness of automatic processing, since apart from rudimentary transformations on the signal itself (such as amplification), any processing must first involve *interpreting* the signal into a symbolic representation. Such signal-to-symbol conversion lies in the domain of "artificial intelligence" and is presently unreliable. It is beyond the scope of this paper, which deals with symbolic representations.

Users of applications that prepare documents are seldom aware of whether or not some aspects of their information are being stored as measurements rather than codes, and of the significance of this. This is particularly true given the prevalence of WYSIWYG (what you see is what you get) and direct manipulation in user interfaces, since these make signals and symbols indistinguishable. Therefore, users may find it arbitrary that certain processing operations can be performed automatically on certain documents but not on others.

3.2 Unix tools

Many command-line tools commonly associated with the Unix environment are suitable for performing transformation operations on text-based data. Bentley [5] describes the task-specific languages of these tools as "little languages"; several are reviewed in Salus [35]. Three themes often recur: regular expressions, pipes, and preprocessors.

3.2.1 Regular expressions The non-interactive stream editor, `sed`, can perform regular-expression based substitution operations on its input, and has a command language similar to that of the interactive editors `ed` and `vi`. This language is Turing-powerful—a Turing machine can be emulated by a set of search-and-replace commands in a loop—and `sed` scripts exist for numerous transformation tasks.

Since `sed` scripts are cryptic and can be difficult to maintain, the support for regular expressions and other text processing has also become a major feature of some scripting languages with higher-level constructs, such as Awk, Perl, Ruby, Python, and Emacs lisp, which are frequently used to achieve transformations.

From 1962 to 1967, Bell Labs developed the String-Oriented Symbolic Language, or SNOBOL [15]. This utilises a language for pattern matching which is considered by some to be clearer than regular expressions as well as being more powerful; it supports recursively-defined patterns and is in some respects similar to a parser generator (see Section 3.3) with full backtracking. At present, it is rarely used due to its relative slowness and the consequent prevalence of regular-expression based systems. In comparison with regular expressions, SNOBOL code tends to use fewer special characters and a higher-level structure.

3.2.2 Pipes Unix shells, such as `sh`, `bash`, `zsh`, `ksh`, `csh`, `ash`, `tcsh` etc, support various control-flow features and allow the construction of pipelines to pass the output of one command to the input of another, perhaps with other commands acting as "filters" to perform transformations on data while it is in the pipeline. Hence complex transformations can be built up from smaller primitives. The Unix environment provides many commands that can be used in pipelines to obtain such functionality as sorting, searching, and arithmetic evaluation. It is also possible to construct pipelines graphically (see, for example, Spinellis [36]).

3.2.3 Preprocessors and macros A preprocessor copies its input to its output, and, while doing so, it checks for certain embedded codes and acts on them. Generally, the embedded code that the preprocessor recognises is interpreted, executed, and replaced with its output. This is termed "preprocessing", because the output produced is frequently passed to another program, such as a compiler or a typesetting package, for further processing and display.

When considering a preprocessor's role in the conversion of notations, it is useful to differentiate between the input of the *transformation* and the input of the *preprocessor*. Preprocessors can be thought of as having two distinct modes of operation:

1. The input of the preprocessor is fixed, and encodes the nature of the transformation. The input of the transformation is provided separately, in the form of the preprocessor's configuration or environment, and is queried by the embedded code.
2. The input of the preprocessor is the input of the transformation. The nature of the transformation is governed by the configuration of the preprocessor or by including a library of code and definitions.

The fixed input mode is used when the overall form of the desired output is fixed (does not depend on the transformation's input), although some parts of the output do depend on the transformation's input data. Data-dependent code is embedded into a fixed document. This is the approach taken by server-side embedded scripting languages on Web servers, and by the C preprocessor when it is used to customise some code according to the user's desired configuration.

A well-known example of server-side scripting is PHP. Server-side scripting systems are also available for several existing programming languages including Python, Java, Tcl, Perl and Scheme, as well as a few specially-designed languages.

The user-supplied input mode is used to accomplish more complex tasks. The macro language built in to the typesetting software TEX [23], for example, has enabled other languages to be transformed into plain TEX for typesetting. This includes the LATEX language for structured documents, MusiXTEX [38] for musical notation, and XML formatting objects via PassiveTEX [31]. Sometimes the macros redefine the input language completely, but usually the two languages can be mixed and the preprocessor effectively adds new features to the output language.

3.3 Parser generators

A parser is a program that takes a stream of input and a grammar, and uses the grammar to calculate the parse tree (the hierarchical structure) of the input. Actions are taken on the resulting structure. This can be used as the basis of a transformation. Since much of the effort in creating a parser can be automated by using a parser generator, or "compiler compiler" [1], this method can be useful when implementing some transformations, particularly those that follow a complex input structure. Attribute grammars [22] are frequently used for such a purpose, by recursively working on the parse tree in a bottom-up fashion. The contents of terminal nodes in the tree are used to calculate the attributes of non-terminal nodes, which in turn influence the attributes of higher non-terminals until it is possible to take action on a non-terminal and its attributes. It is however necessary to write the "action" code in a general-purpose programming language that is supported by the parser generator, but for some transformations this might be as simple as printing out the parsed data in a different order.

Perhaps the most famous example of a parser generator is the Unix tool Yacc (Yet Another Compiler Compiler) and its GNU equivalent, Bison. These require auxiliary code to *tokenise* the input, i.e., to group the characters into units such as integers and identifiers. Normally, the associated tool `lex`, which is based on regular expressions, is used as a tokeniser. These tools are designed primarily for speed. Creating fast parsers is desirable when they will be used in production compilers, as these may be given large quantities of code. However, in the context of transforming notations for individual needs and requirements, the speed of the parser might be less important than the usability of the parser generator. If there is less scope for error when writing the grammar for the parser generator, then transformations can be prototyped more quickly and with less effort and resources. In many cases, this will outweigh the longer runtimes of less-efficient parsers.

GLR (generalised look-right) parsers, such as Elkhound [27] and more recent versions of GNU Bison, help to relieve the problem by supporting arbitrary grammars, using backtracking if necessary. A separate tokeniser is not required, as tokens can be included literally in the grammar, hence reducing the level of skill that is required on the part of the parser generator's user. It is still necessary, though, for the user to have some understanding of parsing theory and to avoid certain pitfalls. For example, many parser generators will

crash (or produce parsers that crash) if a user specifies an action at the beginning of a left-recursive reduction rule.

3.4 Rewriting systems

Rewriting theory, sometimes also called "rule-based programming", is often used in equational reasoning, including automated deduction, automated verification of specifications, type theory, etc. A rewriting system involves a collection of "rewriting rules", which are directed equations. Informally, they specify that whenever a given pattern of symbols is encountered, it must be rewritten in a specified form. These rules are applied in sequence in accordance with a *rewriting strategy*, which is usually a *normalising* strategy, meaning that the structure is repeatedly transformed until it is in "normal" form and the rules cannot effect further changes. Rewriting is Turing-powerful.

The difference between rewriting and simple "search and replace" (with or without regular expressions) is that rewriting operates on hierarchical structures, rather than on an unstructured string of characters. A rewriting system operates on data that has already been parsed. Hence, it is possible to create rules that specify such things as what *types* of data the rule applies to and in what context, possibly with other conditions added. Rule patterns frequently correspond to fragments of the parse tree. The resulting rules tend to be more concise than their search-and-replace counterparts. Rewriting strategies sometimes specify that the pattern replacement must first occur in a particular part of the parse tree, such as the outermost level.

TXL [9] is an example of a generic rewriting language that incorporates a parser generator, allowing arbitrary languages to be parsed, and subsequently applies rewrite rules to the abstract syntax trees that are generated. The commercial DMS software maintenance system [4] employs a similar method for its program transformations.

The Stratego language [19] also uses rewrite rules on abstract syntax trees. In Stratego, the rewriting strategy is user-definable. Stratego itself does not include facilities to parse and format the syntax trees, the latter being provided by auxiliary tools such as XT [20].

Rewriting languages are also used by the well-known symbol-based mathematics package Mathematica [43] and the Rigal programming language [3], and logic programming languages such as Prolog are also based on it.

Rewriting is best suited for mathematical structures where the rewriting rules follow naturally from the mathematical definition of the structure. Any transformation that can be expressed informally as a set of "this pattern should be re-written as that" statements, which capture the complete transformation and are not merely examples of it, is likely to be easily implemented in a rewriting system as long as the input data can be parsed into the system. Rewriting systems may be more difficult to use in cases where it is less obvious what the rules should be, or when the transformation needs to go through one or more intermediate states before the desired result can be achieved (this needs more thought on the part of the transformation programmer).

3.5 XML-based tools

There is no shortage of books and websites on XML [45] and related presentation tools, many of which make use of the XSLT transformation language [44]. This is effectively a variation on the rewriting systems mentioned above, except that it is not necessary to construct a specialised parser for each type of input before the rewriting can begin, because well-written XML makes the relevant structure explicit. Cascading style sheets (CSS) provides a more limited way of transforming XML text for presentation (CSS level 2 and above can be used with arbitrary XML).

XML and XSLT can be verbose. As with other frameworks, some transformation tasks require a lot of code, and this can present problems, particularly for people with print disabilities, due to the overhead of writing and navigating the code. This problem can be partly alleviated by XML-aware editors and other development tools, as long as these applications themselves are accessible.

3.6 Multiple hierarchies and matrix-like structure

Most symbolic data is hierarchical, or *tree-like*, at least after it has been parsed. Generalised markup languages, such as XML, can be used for describing hierarchical structures over documents and data directly. In generalised markup languages, a piece of data can be enclosed within an "element", which can in turn be a member of a higher-level element, etc.

It is often overlooked that much data is also *matrix-like* in nature, that is, it can be indexed along two or more orthogonal dimensions which can be addressed independently. Tables and spreadsheets are matrices. A musical score (which represents parallel streams of events) is matrix-like, and so are parallel translations of literary works. They can be interpreted as having tree-like structures, but there are several equally-valid branching orders. Often it is possible to read a document in several different ways, using, in effect, several different methods of indexing into the items of data that make up the document. It can be useful to treat these indices as the different dimensions of a multi-dimensional matrix, so that switching from one system to another amounts to slicing along a different dimension.

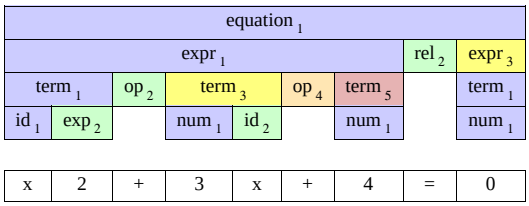

Fig. 2 4DML's representation of a parse tree

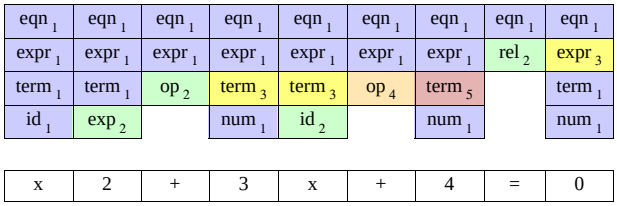

Fig. 3 The true pointset in Figure 2

The programming language APL uses matrices as a primitive type. However, it is often more useful to employ a conventional tree-like hierarchical structure, particularly if the data's structure is recursive, as is the case with mathematical expressions, which can contain other expressions to any depth. However, using a hierarchical structure throughout makes things more complex when they would perhaps be better represented as matrices. When a single hierarchical structure is not the most natural way to represent the structure of a notation, it can impose artificial restrictions on the notation's transformation [7].

Multiple overlapping structures can be represented in a single hierarchy by making use of such things as XML linking (ID and IDREF), but this can be complex and require more effort from the programmer. The simultaneous handling of hierarchical structures and matrix-like or overlapping structures can be viewed as a challenge for many existing transformation frameworks.

4 4DML

4DML (four-dimensional markup language) is a generalised transformation framework developed by the first author [7]. It represents data in such a way that it can be treated as either tree-like or matrix-like, or both, as appropriate. This orthogonality also allows multiple, independent hierarchies over the same data without undue complexity. This is useful when there is more than one set of markup, e.g., logical divisions that are separate from physical divisions and indexing divisions.

4.1 The data structure

4DML's data structue is illustrated in Figure 2. It consists of symbol-based data and a hierarchical structure over that data that indicates how the original input was parsed. However, it is possible to add multiple hierarchies that are orthogonal to each other by adding more layers to the diagram.

The illustration corresponds to a four-dimensional pointset: the horizontal dimension corresponds to which symbol is being marked up, the vertical dimension to the depth of the markup, the text to the type of markup, and the numbers (which are also represented by background colours) to the position of each piece of markup among its peers. That such positioning information is needed is indicated by Figure 3, in which the true pointset is shown. All markup is duplicated such that the markup over any particular symbol can be represented out of context.

The left-to-right order is irrelevant (it can be reconstructed from the other data), and is shown for clarity only. Since different notations may represent the same information in different arrangements, the ordering should be a property of the markup, not of the underlying data.

4.2 Transformation by model

The primary algorithm associated with 4DML is "transformation by model", which takes some 4DML input along with a "model" of the desired structure, and transforms the input as necessary to reflect the structure of the model. The algorithm can also report which objects have been lost in the process, if any.

The algorithm works by performing a top-down traversal of the model, and, while doing so, reads off relevant parts of the input. Since 4DML can be read in many different ways, it is not difficult to read it in whatever way is dictated by the structure of the model, regardless of whether or not this matches the original structure. Hence, it is possible to perform complex structural transformations merely by writing down the form of the desired result. It is also possible to follow the structure of the input in the same way as a conventional stylesheet processor.

In other words, each element in the model will cause the occurrence of the following:

1. The input is searched for all elements that match the name of the model element. Only such elements at the highest level at which they occur will be used. The search will cut across all other markup.
2. The input is divided into groups, one for each distinct element that was found, and the groups are sorted by position number. Any other markup from the input is included in each group.
3. Any model code that occurs within the current model element is executed once for each group.
4. If the model element is empty (a leaf node), then the data from each group is copied to the output, discarding all remaining markup.

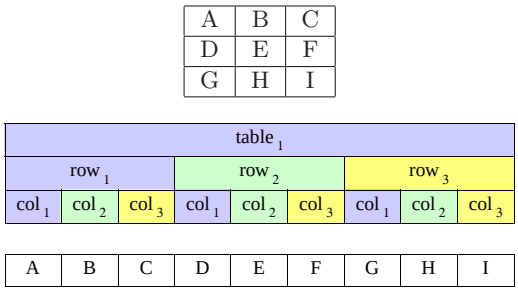

Fig. 4 A 3×3 table and an illustration of how it might be represented in 4DML

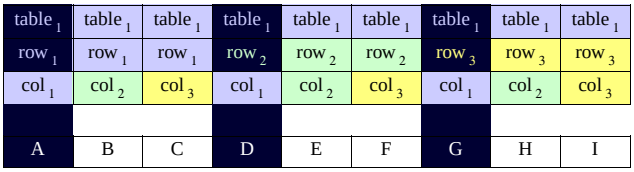

Fig. 5 Selecting column 1

Thus, any element X in the model is effectively interpreted as universally quantified ("for each X"). Arbitrary text in the model is copied to the output whenever it is encountered. Model elements can be given attributes (parameters) to specify their behaviour in a flexible manner.

Figure 4 shows how a 3×3 table or matrix, with the input format representing columns within rows (as is the case with HTML) might be represented in 4DML. It is equally easy to select a row as to select a column (Figure 5), so if the user wishes to transform the table into a notation that requires it to be read in columns rather than rows then this is virtually transparent.

The complete specification of the order in which the transposed table is to be written is essentially the nested statement:

- For each "col", select that "col" and
  - For each "row", select that "row" and
    - Write out what is selected

In 4DML's compact model language (CML), this is abbreviated to `col/row` (more correctly, `table/col/row`, which would properly handle the case of several tables), with appropriate parameters added so that the markup is re-named or deleted as required by the typesetting or display software that will produce the output notation.

4.2.1 Compact Model Language (CML) CML is a text-based language designed to assist with the coding of 4DML models. In practice, most models have a tail-recursive structure and express such things as "for each song, for each verse, for each syllable,..." which in XML would require a number of closing tags:

```
<SONG> <VERSE> <SYLLABLE>
</SYLLABLE> </VERSE> </SONG>
```

In CML, the above is expressed as `SONG/VERSE/SYLLABLE`. CML also has other operators and can represent any hierarchical document, but its syntax is specifically designed for representing typical 4DML models concisely.

Complete models in CML are often small enough to be given to the processor as command-line arguments. Alternatively, CML can be embedded into a file as a macro language (similarly to PHP), which is useful when there is a large amount of text (such as typesetting markup) to include before, after, or between the input.

CML's syntax resembles that of XPATH in XSLT, but its working is fundamentally different. CML expresses the form of the desired output, whereas XPATH expresses a path to be taken through the input. With CML, it does not matter if the elements of the input are nested in a different way than is shown by the CML model, since 4DML transposes the nesting as appropriate. Hence, the author of the model is encouraged to consider the structure of the result rather than following the structure of the input.

4.3 Matrix markup language (MML)

Before 4DML can work with structured data, the data must first be made available to it. It can be cumbersome to hand-code matrix-like data in a hierarchical markup language like XML, since the markup is very verbose and repetitive. For example, in coding the lyrics of a song, one might have to enclose each syllable in a `<SYLLABLE>...</SYLLABLE>` pair, whereas it would be easier to define a separator (for example, the whitespace) to stand for "next syllable" (other separators can advance the verse number or the translation).

In the general case, one can construct a parser for an arbitrary domain-specific input language, but this can require a significant amount of effort for an end-user. There is scope for a markup language that provides for some simple re-definitions (such as "whitespace means next syllable") while not being as complex as a parser generator. Matrix Markup Language (MML) is a text-based language designed to assist with the coding of 4DML data in this way. Arbitrary strings can be defined to advance markup at various levels of the hierarchy (the precedence is shown by the order in which they are specified), and multiple independent hierarchies can also be expressed.

The 4DML prototype can also process XML input.

5 Example applications of 4DML

This section illustrates some example cases where 4DML can be useful for converting between different notation

systems. All of these tasks can be achieved without the use of 4DML, but would require several different transformational frameworks and tools to achieve the same results, and the code could in some cases be complex and difficult to maintain.

This paper does not address the details of 4DML or the precise operation of its models, but some example models are illustrated and briefly explained.

5.1 Website "scraping"

Web services that publish content such as weather reports are often very detailed. A weather forecast might contain a tabulation of details on air pressure, temperature, wind speed and pollen counts, over several days and perhaps in several locations. For those relying on large print, speech synthesis or Braille, it can take much time to locate the small amount of data that is actually required, particularly if the software cannot guess the most logical way of reading the table.

The practice of "scraping" refers to the use of an automatic program to interpret data that is presented with the intention of being read by a human, such as data from a screen display or a complex website. Historically, "scraping" has frequently been used in the area of disabled users' access to computing. Its main disadvantage is that any re-design in the layout of the screen or website is likely to break the program that reads it, so these programs need frequent maintenance.

4DML has been used as a "website scraping" system to read off appropriate parts of an HTML weather forecast, providing a short summary such as:

> Saturday: Sunny Intervals. Sunday: Light Showers. Monday: Cloudy.

The transformation is essentially a matrix transposition, so that the website's table is read vertically by column, and a clipping (a limit on the range of data that is output). Both of these are expressed in a concise transformation "model":

```
td start-at=2 end-at=4 between=". "/tr start-at=2
end-at=3 merge/(font, ": ", alt)
```

Informally, this means: "For columns (tds) 2 through 4, do the following (while outputting . between each column): For the merged content of rows (trs) 2 and 3, write out the text enclosed by font, then :, then the text enclosed by alt." This happens to be the day name and the weather forecast respectively, which are not explicitly labelled. Merging is necessary because the font and alt are in different rows. An alternative approach would be to state:

```
font after=": ", alt
```

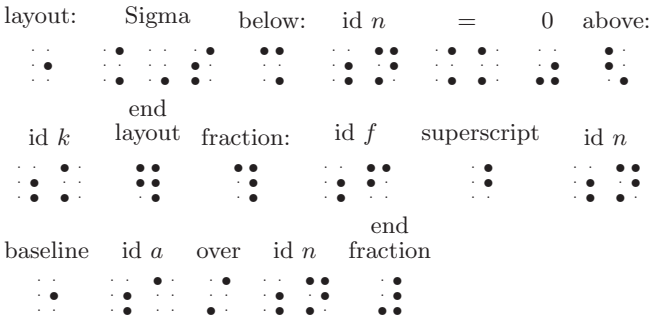

Fig. 6 Annotated Braille mathematics (Nemeth linear code)

This model occasionally needs to be re-written to cope with changes in the site's layout, although not every layout change affects the model. The script has been in daily use for many months.

If the website were to release weather data in a standardised format that is specifically intended for processing by a program, then this would remove the need to re-write the model whenever the web designers change the layout. However, it is often the case (at the time of writing) that data in such formats is only available for a fee, since it would make it easier to set up a competing source of weather forecasts.

5.2 Mathematics reading

4DML was used to parse a MathML document and output the result as text suitable for a speech synthesiser, appropriately rendering the included mathematical expressions. For example, the expression

$$\sum_{n=0}^{k} \frac{f^n a}{n}$$

was transformed into "sigma from n equals 0 to k of f to the n a over n". The model for this is somewhat larger because of the number of different mathematical symbols it needs to translate.

By changing the model, Braille output was produced, either as an annotated visual representation (Figure 6) or as codes suitable for controlling an automated Braille embosser or a Braille display.

The inclusion of mathematics as an example is merely illustrative, because mathematics can already be transformed and customised using specialised systems such as AsTeR [32]. The novelty of 4DML is better shown in other domains, but we felt it was important for any new transformation system to show that it can process mathematics as well.

5.3 Typesetting for language learning

When teaching written Chinese to Western students, Chinese characters are often written with small pronun-

ciation guides around them, and sometimes the meaning of each character or group of characters is indicated alongside in the language of the student.

There are several systems for indicating pronunciation. Some use special phonetic symbols (such as *zhuyin*, which is informally known as bopomofu), and others employ the Latin alphabet. These *pinyin* (alphabetical) systems use either numbers or special accents to indicate tonal inflection, and they use various different spellings that are not always intuitive to native speakers of English (this is a common source of pronunciation errors). Different students prefer different *pinyin* systems, and many beginners prefer to devise their own systems for private use, but there are advantages in learning one of the standardised systems (such as *Hanyu Pinyin* which is used in mainland China) since it is widely used in printed books, dictionaries, and computer software.

Many students try to move from their private notations onto the standard that they wish to learn. Such moving is usually done by writing both notations in parallel and progressively deleting parts of the private notation according to the student's progress. In this way students can be guided from one *pinyin* system to another, and also towards reading Chinese characters directly.

The "ruby" system of writing pinyin above the characters is difficult for partially-sighted students because it employs very small print, and if it is enlarged without adjustment then the result can be unwieldy and give rise to *tracking* problems (i.e., people with certain sight conditions lose track of which line they are reading). Additionally, if these students want to produce a customised version of the notation for themselves, then they are impeded by the graphical way in which the symbols are positioned relative to each other; they either have to write it by hand or use a graphical wordprocessor, and both can be difficult for visually-impaired students.

4DML allows a user to produce a "model" of the desired output in the language of a Chinese-capable, text-driven typesetting package such as CJK–LaTeX. In Figure 7, the different lines of text, which have been printed at similar sizes to facilitate zooming, have been brought very close together so as to aid tracking, and colour has been used to compensate for the resulting crowding and to further assist with tracking (shades of grey can also be effective). The blue private notation (which has been generated automatically) has mostly been taken away, and the few Chinese characters that the user has learned have been duplicated into the pinyin line. Syllable separation dots (which are not standard in Chinese) have also been used, as have parentheses to indicate a higher level of grouping. It is also possible to include symbols denoting gestures (if giving a presentation) or musical tones (if singing a song) according to the student's needs.

Adjustments to the model can be made over time, such as changing the fonts, spacing and colours, changing the order in which the lines appear (or deleting some lines altogether), and controlling which Chinese characters (and which aspects of pinyin) are assumed to be known. Sometimes it is desirable to produce several different presentations of the same text, as when other students or a teacher is involved or when more space is needed for handwritten corrections, and sometimes it is desirable to send the output as plain text to a Chinese-capable email client. These are achieved by using different models.

Fig. 7 A customised notation to assist with Chinese studies

(a) CEDICT format:

```
我 [Wo3] /I/
试图 [shi4 tu2] /attempt/
说 [shuo1] /speak/
中文 [zhong1 wen2] /Chinese/
```

(b) PinYin-based format:

```
Wo3 shi4-tu2=attempt shuo1 zhong1-wen2
```

(c) Song lyrics format:

```
Shang4 di4 xian4 yi3 xia4 ling4, pai4 qian3
上 帝 现 已 下 令, 派 遣
Shang4 di4 di4 shang4 zi3 min2, wang2 guo2
上 帝 地 上 子 民, 王 国

guang1 zhao4 lin2 di4 shang4 ...
光 照 临 地 上 ...
qin2 xuan1 jiang3; an1 wei4 ...
勤 宣 讲; 安 慰 ...
```

Fig. 8 Different ways of inputting annotated Chinese

There are many established ways of inputting ideographic characters. Most of these involve typing a character's pronunciation (in some dialect) or codes that represent clues about its appearance, and then selecting from the characters that match those criteria. This essentially amounts to making queries on a database or *dictionary* of characters, and for the language student the system can be extended as follows:

1. Use a dictionary that includes definitions in English or another language, and allow searches on these definitions *as well as* any of the character's other attributes (using this method it is also possible to retrieve a sequence of several characters in one operation);

2. Once the character(s) have been identified, insert into the document the entire dictionary entry (including pronunciation and definition), not just the character itself.

This means that the input to the 4DML transformation is a sequence of dictionary entries which can be edited for readability (see Figure 8). The user can either use a specially-programmed input method or can construct the sequence manually by searching a dictionary file and copying from it (perhaps with the aid of keyboard macros).

If Chinese characters have already been provided, then it can be unclear how they should be grouped into words (which can be formed of two or more characters), and in this case it is useful to keep the pinyin together so that its grouping can be changed quickly (for example, by replacing a space with a hyphen). Sometimes Chinese characters may not be available at all, and only pinyin and English (or another language) are present.

If the input has been provided by others, then it may be in a more esoteric format, such as song lyrics, where a line of pinyin is placed above a line of characters, and pairs of lines from different stanzas are interleaved.

5.4 Aspect-oriented music encoding

There are essentially four ways of inputting Western musical notation into a computer:

1. Scanning. This is only feasible if the visual rendering of the original musical notation is of sufficiently good quality and is rarely possible with handwritten music.
2. Using a *direct manipulation* music publishing system such as *Sibelius* [11]. Such software can be difficult to use for people with print disabilities.
3. Playing the music on a keyboard or similar. Due to the limitations in artificial "aural skills", the resulting notation is usually inaccurate for all but the simplest music.
4. Writing in a "musical code" (also called "little music language"), i.e., a special computer language that gives instructions to a music typesetting program.

Since the latter option is the most feasible for print-disabled people, that is the one that is examined here.

Modern Western musical notation has a large "vocabulary" of possible symbols, so the computer languages that represent it have to be fairly complex. This means that those who wish to write in such languages will either have to spend time learning them (which can be too much for an occasional user), or will be slowed down by the frequent need to refer to a reference manual or online help system. If the user is also the composer, then this problem might restrict the composition.

The first author's approach to addressing this problem is to pass through the music several times, each time

```
begin music
begin part
!block pitch
have whitespace character as bar note

r rrrd ddddfca aarrd ddddfca aadce gfcdfeca gfcddfeeg
gfbagffg dcfffg feaabb ddcc bbaa gfgaabaadfa
!endblock

!block duration
have whitespace character as bar note

0 2488 8881144 28888
8881144 28114 11481148
114111148 11481148 114848
114848 4882 4882 88888883333
!endblock

!block accidental
have whitespace character . as bar note rubbish

. . .....s . .....s ...s ..s ..s ......s .sn.s .....n
..n nn .s...n...n
!endblock
```

Fig. 9 Extract from a musical score using one method of aspect-oriented music encoding. In this case the other aspects include *octave*, *dot*, *tuplet*, *tie*, *articulation*, *dynamics*, *keychange* and *text*.

encoding just one or two aspects (note letters, octaves, enharmonics, note values, dots, tuplets, phrasing, articulation, etc). Thus the user can type in *all* of the note letters in the piece, then go back to the beginning and type in *all* of the octaves, and so on (see Figure 9). This is more efficient because only a small amount of vocabulary needs to be considered at any one time. It also introduces the possibility of distributing the work-load between several people with limited training.

Some modern styles of composition can also lend themselves to an aspect-oriented construction. The independent aspects of the notation that are progressively added are not necessarily the aspects of musical notation; rather, they are the aspects of the compositional framework defined by the composer, which is then converted into standard musical notation by the 4DML model. The underlying idea is similar to the separation of concerns in aspect-oriented programming [10].

4DML was used to transform between a variety of aspect-oriented formats and the languages of three music typesetting systems. A different model was used to produce Japanese Koto tablature (see Figure 10), and it is also possible to convert into several different versions of Braille music and other notations as required. This is discussed in more detail elsewhere [8].

Fig. 10 Japanese Koto notation produced using 4DML with Lout and CJK–LaTeX

6 Availability

The 4DML system described in this paper has been implemented and is available for download. We aim to improve the quality of the distribution shortly. It is in regular use for a variety of tasks by at least one individual with low vision, and it has been used by others although it has not yet been possible to obtain feedback from large numbers of users.

7 Conclusion

Virtually all information-society applications involve notations [29, 13], and the transformation of these between different versions is a component part of universal access, since it can help to cater for individual user requirements and for different tasks and environments. Tools that support the programming of such transformations can make it easier to create new notations on demand and to implement universal design. They are not in themselves complete solutions, but can contribute towards such an overall objective.

The 4DML system allows the user to specify the structure of the desired result in a fairly concise manner, without needing to design an algorithm or a set of transformation rules as is normally the case. This can help developers, particularly print-disabled developers (of which the first author is an example), to prototype transformations more quickly and to experiment with new notations during efforts to address educational needs.

Acknowledgements The first author is supported by a studentship from the UK's Engineering and Physical Sciences Research Council (EPSRC).

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. Jonathan Allen. Reading machines for the blind: The technical problems and the methods adopted for their solution. *IEEE Transactions on Audio and Electroacoustics*, 21(3):259–264, June 1973.
3. Mikhail Auguston. RIGAL—a programming language for compiler writing. *Lecture Notes in Computer Science*, 502:529–564, 1991.
4. Ira D. Baxter. DMS: practical code generation and enhancement by program transformation. In *Workshop on Generative Programming*, pages 19–20, 2002.
5. Jon Louis Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.
6. Silas S. Brown and Peter Robinson. A World Wide Web mediator for users with low vision. In *ACM CHI 2001 Workshop No. 14*. `http://www.ics.forth.gr/proj/at-hci/chi2001/files/brown.pdf`.
7. S.S. Brown and P. Robinson. Automatically rearranging structured data for customised special-needs presentations. In Simeon Keates, P. John Clarkson, Patrick Langdon, and Peter Robinson, editors, *Universal Access and Assistive Technology: proceedings of the Cambridge Workshop on UA and AT*, pages 109–118, Mar 2002.
8. S.S. Brown and P. Robinson. Transforming musical notations for universal access to performance and composition. In Simeon Keates, John Clarkson, Patrick Langdon, and Peter Robinson, editors, *Designing a More Inclusive World: proceedings of the Cambridge Workshop on UA and AT*, pages 123–132. Springer, Mar 2004.
9. James R. Cordy, Charles D. Halpern, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceedings of The International Conference of Computer Languages*, pages 280–285, Miami, FL, Oct 1988.
10. Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, Oct 2001.
11. Ben Finn and Jonathan Finn. *Sibelius: The Music Notation Software*, 2001. Sibelius Software Ltd, Cambridge, `http://www.sibelius-software.com/`.
12. Office for Standards in Education. *Inspection Report—RNIB New College, Worcester*, page 37. Alexandra House, 33 Kingsway, London, WC2B 6SE, Oct 2000. Inspection number 223644.

13. T. R. G. Green and A. F. Blackwell. Design for usability using cognitive dimensions. Tutorial session at British Computer Society conference on Human Computer Interaction HCI'98, 1998.
14. Peter Gregor and Alan F. Newell. An emperical investigation of ways in which some of the problems encountered by some dyslexics may be alleviated using computer techniques. In *Proceedings of the Fourth International ACM Conference on Assistive Technologies ASSETS 2000*, pages 85–91, Nov 2000.
15. R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*. Prentice-Hall, second edition, 1971.
16. Dirk Hermsdorf, Henrike Gappa, and Michael Pieper. Webadapter: A prototype of a WWW-browser with new special needs adaptations. In *Proceedings of the 4th ERCIM Workshop on 'User Interfaces for All'*, number 8 in Long Papers: WWW Browsers for All, page 15. ERCIM, 1998.
17. Julie A. Jacko, Max A. Dixon, Robert H. Rosa, Jr., Ingrid U. Scott, and Charles J. Pappas. Visual profiles: A critical component of universal access. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 1 of *Profiles, Notes, and Surfaces*, pages 330–337, 1999.
18. Julie A. Jacko and Andrew Sears. Designing interfaces for an overlooked user group: Considering the visual profiles of partially sighted users. In *Third Annual ACM Conference on Assistive Technologies*, pages 75–77, 1998.
19. Patricia Johann and Eelco Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):1–34, 2000.
20. M. Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. *Electronic Notes in Theoretical Computer Science*, 44, 2001.
21. Mitsuji Kadota. *Japanese Braille Tutorial*, Oct 1997. `http://buri.sfc.keio.ac.jp/access/arc/NetBraille/etc/brttrl.html`.
22. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–146, 1968.
23. Donald E. Knuth. *The TEXbook*. Computers and Typesetting. Addison-Wesley, 1986.
24. Didier Langolff, Nadine Jessel, and Danny Levy. MFB (music for the blind): A software able to transcribe and create musical scores into braille and to be used by blind persons. In *Proceedings of the 6th ERCIM Workshop on 'User Interfaces for All'*, number 17 in Short Papers, page 6. ERCIM, 2000.
25. S. Ludi. Are we addressing the right issues? Meeting the interface needs of computer users with low vision. In Simeon Keates, P. John Clarkson, Patrick Langdon, and Peter Robinson, editors, *Proceedings of the First Cambridge Workshop on Universal Access and Assistive Technology*, pages 9–12. Engineering Design Centre, Cambridge University Engineering Department, Mar 2002. Technical Report 117, ISSN 0963–5432.
26. Bill McCann. *GOODFEEL Braille Music Translator*, Jun 1997. Dancing Dots Braille Music Technology, `http://www.dancingdots.com/`.
27. Scott McPeak. Elkhound: A fast, practical GLR parser generator. Technical Report UCB/CSD-2-1214, University of California, Berkeley, Computer Science Division (EECS), University of California, Berkeley, California 94720, Dec 2002.
28. Elizabeth D. Mynatt and W. Keith Edwards. Mapping GUIs to auditory interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Audio and Asynchronous Services, pages 61–70, 1992.
29. Cognitive Dimensions of Notations. T. R. G. Green. In Alistair Sutcliffe and Linda Macaulay, editors, *People and Computers V: Proceedings of the Fifth Conference of the British Computer Society*, pages 443–460. Cambridge University Press, Nov 1989.
30. Ian J. Pitt and Alistair D. N. Edwards. Improving the usability of speech-based interfaces for blind users. In *Second Annual ACM Conference on Assistive Technologies*, Vision Impairments – II, pages 124–130, 1996.
31. Sebastian Rahtz. PassiveTEX. Text Encoding Initiative. Available in most TEX distributions, 2003.
32. T. V. Raman. *Audio System for Technical Readings*. PhD thesis, Cornell University, 1994.
33. T. V. Raman. Emacspeak: a speech-enabling interface. *Dr. Dobb's Journal*, Sep 1997.
34. Richard Rubinstein and Julian Feldman. A controller for a Braille terminal. *Communications of the ACM*, 15(9):841–842, September 1972.
35. Peter Salus, editor. *Little Languages and Tools*, volume 3 of *Handbook of Programming Languages*. Macmillan Technical, first edition, 1998.
36. Diomidis Spinellis. Unix tools as visual programming components in a GUI-builder environment. *Software—Practice and Experience*, 32(1):57–71, January 2002.
37. Constantine Stephanidis. Aims and scope. *Universal Access in the Information Society*, 1(1):A4, 2001.
38. Daniel Taupin, Ross Mitchell, and Andreas Egler. *MusiXTEX: Using TEX to write polyphonic or instrumental music*, Apr 1999. `ftp://ftp.gmd.de/music/musixtex/musixdoc.ps`.
39. David Taylor and Christopher Harris. About nystagmus. Technical report, Nystagmus Network, 108c Warner Road, Camberwell, London, SE5 9HQ, UK, Sep 1999. `http://www.btinternet.com/~lynest/nystag.pdf`.
40. Jim Thatcher. Screen reader/2: Access to OS/2 and the graphical user interface. In *First Annual ACM Conference on Assistive Technologies*, Vision Impairments – I, pages 39–46, 1994.
41. Jennifer Watts-Perotti and David D. Woods. How experienced users avoid getting lost in large display networks. *International Journal of Human-Computer Interaction*, 11(4):269–300, 1999. ISSN 1044–7318.
42. G. Weber, D. Kochanek, C. Stephanidis, and G. Homatas. Access by blind people to interaction objects in MS Windows. In *Proceedings of the ECART 2 European Conference oon the Advancement of Rehabilitation Technology (Stockholm)*, page 2, May 1993.
43. Stephen Wolfram. *The Mathematica Book*. Cambridge University Press, fourth edition, Apr 1999.
44. World Wide Web Consortium. *XSL Transformations (XSLT) Version 1.0, W3C Recommendation*, Nov 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.
45. World Wide Web Consortium. *Extensible Markup Language (XML) Version 1.0 (Second Edition)*, Oct 2000. `http://www.w3c.org/TR/2000/REC-xml-20001006`.

46. Mary Zajicek, Chris Powell, and Chris Reeves. A web navigation tool for the blind. In *Third Annual ACM Conference on Assistive Technologies*, pages 204–206, 1998.