INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the original text directly from the copy submitted. Thus, some dissertation copies are in typewriter face, while others may be from a computer printer.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyrighted material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is available as one exposure on a standard 35 mm slide or as a 17" × 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6" × 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Assuring accessibility of complex software systems

Nicholas, Charles Kenneth, Ph.D.

The Ohio State University, 1988

Copyright ©1988 by Nicholas, Charles Kenneth. All rights reserved.
ASSURING ACCESSIBILITY OF COMPLEX SOFTWARE SYSTEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

BY

Charles Kenneth Nicholas, B.S., M.S.

****

The Ohio State University

1988

Dissertation Committee:  Approved By

Dr. Sandra A. Mamrak

Dr. Douglas S. Kerr

Dr. Stuart H. Zweben

Adviser

Department of Computer and Information Science
Dedicated to Sue
Acknowledgments

I want to express my thanks to my advisor, Sandy Mamrak, for her help and encouragement over these last several years. Thanks also go to the members of my reading committee, Doug Kerr and Stu Zweben, for their comments and suggestions.

It has been my pleasure to be a member of the Chameleon research group for the last few years, and I am grateful to the other members for many helpful discussions. Special thanks go to Mike Share and Julie Barnes for reading earlier versions of this thesis.

Finally, I want to extend my profound thanks to my parents, and to my wife Susan for their loving support throughout a relatively lengthy graduate career.

This work was supported in part by the Applied Information Technologies Research Center, and by an equipment grant from the Xerox Corporation.
Vita

September 20, 1957 .................. Born - Elyria, Ohio
1979 .................................. B.S. with Honors, University of Michigan
1982 .................................. M.S., The Ohio State University
1982-84 .............................. Researcher, Battelle Columbus Laboratories
1984-present ......................... Graduate Research Associate,
                               The Ohio State University

Publications

"Guest Layering Distributed Processing Support on Local Operating Systems",
with S. Mamrak, P. Maurath, J. Gomez, S. Janardan, 3rd International
Conference on Distributed Computing Systems, Ft. Lauderdale, Florida,

"An Approach to the Solution of Data Conversion Problems in Heterogeneous
Networks," with S. Mamrak and M. Kaelbling, Ohio State University Technical


Fields of Study

Major Field: Computer and Information Science

Studies in:

Heterogeneity in Distributed Computing Systems,
   with Prof. Sandra Mamrak

Software Engineering,
   with Prof. Stuart Zweben

Programming Languages,
   with Prof. William Ogden
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>Vita</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. The Problem: Accessibility of Complex Systems</td>
<td>1</td>
</tr>
<tr>
<td>I.1. Our Contributions</td>
<td>3</td>
</tr>
<tr>
<td>I.2. Organization of this Thesis</td>
<td>4</td>
</tr>
<tr>
<td>II. A Solution: Assuring Accessibility of Software Systems</td>
<td>5</td>
</tr>
<tr>
<td>II.1. The Traditional Approach to User Interface Design</td>
<td>7</td>
</tr>
<tr>
<td>II.2. Our Approach</td>
<td>11</td>
</tr>
<tr>
<td>II.3. The High Level Data Translation Problem</td>
<td>16</td>
</tr>
<tr>
<td>II.4. The Chameleon Translation Architecture</td>
<td>18</td>
</tr>
<tr>
<td>II.5. An Example: The Chameleon Translator Writer</td>
<td>21</td>
</tr>
<tr>
<td>II.6. Summary</td>
<td>23</td>
</tr>
<tr>
<td>III. Related Work</td>
<td>24</td>
</tr>
<tr>
<td>III.1. User Interface Issues</td>
<td>24</td>
</tr>
<tr>
<td>III.1.1. Learn from the User</td>
<td>25</td>
</tr>
<tr>
<td>III.1.2. Build a Prototype</td>
<td>26</td>
</tr>
<tr>
<td>III.1.3. Take Measurements</td>
<td>28</td>
</tr>
<tr>
<td>III.2. Toward Separation of System and User Interface</td>
<td>29</td>
</tr>
<tr>
<td>III.3. Critique of Previous Work</td>
<td>30</td>
</tr>
<tr>
<td>IV. Data Translation and Related Problems</td>
<td>32</td>
</tr>
<tr>
<td>IV.1. A Formal Model of Data Translation</td>
<td>32</td>
</tr>
<tr>
<td>IV.1.1. The Standard Form Approach to Data Translation</td>
<td>32</td>
</tr>
<tr>
<td>IV.1.2. Standard Forms as Braced Languages</td>
<td>33</td>
</tr>
<tr>
<td>IV.1.3. Translation Functions</td>
<td>37</td>
</tr>
<tr>
<td>IV.2. Data Translation Problems</td>
<td>42</td>
</tr>
<tr>
<td>V. The User Conceptual Framework</td>
<td>48</td>
</tr>
<tr>
<td>V.1. Deriving the User Interface</td>
<td>50</td>
</tr>
<tr>
<td>V.2. The User Model and Vocabulary</td>
<td>52</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>V.3. The Conceptual Model</td>
<td>53</td>
</tr>
<tr>
<td>V.4. An MDL Specification</td>
<td>57</td>
</tr>
<tr>
<td>V.4.1. MDL Declarations</td>
<td>58</td>
</tr>
<tr>
<td>V.4.2. MDL and Translation Down</td>
<td>60</td>
</tr>
<tr>
<td>V.4.3. MDL and the Nonstandard Form</td>
<td>65</td>
</tr>
<tr>
<td>V.5. Discussion</td>
<td>66</td>
</tr>
<tr>
<td>VI. The Chameleon Architecture</td>
<td>68</td>
</tr>
<tr>
<td>VI.1. Approach</td>
<td>68</td>
</tr>
<tr>
<td>VI.2. AGs and the Conceptual Model</td>
<td>69</td>
</tr>
<tr>
<td>VI.3. The Six Situations - A System Perspective</td>
<td>71</td>
</tr>
<tr>
<td>VI.4. The Build Phase of the Chameleon Architecture</td>
<td>75</td>
</tr>
<tr>
<td>VII. Using the Translator Writer</td>
<td>79</td>
</tr>
<tr>
<td>VII.1. Connecting the System and the User Interface</td>
<td>79</td>
</tr>
<tr>
<td>VII.2. Using the Translator Writer: A Scenario</td>
<td>81</td>
</tr>
<tr>
<td>VII.3. Using the TW to Build Production Quality Translators</td>
<td>85</td>
</tr>
<tr>
<td>VII.3.1. Experience in Building Translators</td>
<td>85</td>
</tr>
<tr>
<td>VII.3.2. Arguing the Competitiveness of our Approach</td>
<td>88</td>
</tr>
<tr>
<td>VII.3.3. Other Benefits of this Approach</td>
<td>92</td>
</tr>
<tr>
<td>VIII. Conclusions and Future Work</td>
<td>97</td>
</tr>
<tr>
<td>VIII.1. Conclusions</td>
<td>97</td>
</tr>
<tr>
<td>VIII.2. Other Questions</td>
<td>98</td>
</tr>
<tr>
<td>VIII.2.1. The Translator Writer as a Software Tool</td>
<td>98</td>
</tr>
<tr>
<td>VIII.2.2. Possible Extensions or Improvements to Our Approach</td>
<td>100</td>
</tr>
<tr>
<td>Bibliography</td>
<td>102</td>
</tr>
<tr>
<td>Appendix A. Syntax of MDL</td>
<td>107</td>
</tr>
<tr>
<td>Appendix B. An MDL Specification</td>
<td>111</td>
</tr>
</tbody>
</table>
List of Figures

1. The Traditional Approach ............................................................... 8
2. Separating System and User Interface Design .................................. 12
3. Fragment of a Scribe Document ....................................................... 19
4. The Standard Form Approach .......................................................... 34
5. Translation Down as p(h(x)) ............................................................. 40
6. Translation Up as s^{-1}(p^{-1}(y)) ....................................................... 41
7. Three Possibilities for s(x) ................................................................. 44
8. Three Possibilities for s^{-1}(x) ........................................................... 46
9. An MDL Declaration Section ........................................................... 58
10. An MDL Translation Down Specification ........................................ 61
11. Using MDL to Describe the Nonstandard Form ............................... 65
12. The Chameleon Build Phase ............................................................ 76
13. An SGML Document Type Definition ............................................. 77
Chapter I

The Problem: Accessibility of Complex Systems

In this thesis, we present a method for assuring the accessibility of software systems, and an example system developed using this method. Intuitively, an accessibility problem is some aspect of a system that prevents or obstructs a user from doing what he or she wishes. We assume that any system that interacts with a human user does so through a user interface. If the user interface makes it easy for the user to access the desired functionality, then it has done its job. All too often, though, user interfaces place obstacles between the user and the desired functionality. There are several ways in which this can happen, including the use of inappropriate vocabulary, or inappropriate syntax, in the input language. In particular, input language vocabulary and syntax that is oriented towards the system that actually implements the desired functionality is an inappropriate way to give the user access to that functionality.

This use of system-oriented terminology and syntax stems directly from the traditional software development process. In the traditional software development process, a functional specification is created that says what the system is to do. This functional specification is then used to drive the design of the system that implements that functionality. Finally, as an integral part of this system design, the design of the user interface takes place. However, treating user interface design as just another part
of the design of the system makes contamination of the user interface by system-oriented terminology or syntax highly likely, if not inevitable.

To avoid this contamination, we claim that it is necessary for the user interface and the rest of the system to be developed separately. The functional specification can still be used to drive the design of the system that actually implements that functionality. What we are suggesting, and what we demonstrate in this thesis, is:

1. The user interface can be derived directly from the functional specification, and

2. The resulting user interface provides access to the desired functionality.

We demonstrate the viability of this approach by using it in the construction of a user interface for a complicated system that might otherwise have serious accessibility problems. The system used for this demonstration is the build phase of the Chameleon architecture. Chameleon is a system that supports the specification, construction and use of data translation tools. The Chameleon build phase supports the specification and construction of these tools. The architecture uses several advanced concepts from computer science in unconventional ways. As a result, the architecture is rather complicated, and would be inaccessible to its intended users if we didn't take steps to prevent this. We demonstrate the feasibility of our approach by using it in the design of the user interface to the build phase, which is known as the Translator Writer.
1.1 Our Contributions

The point of this research is to show how separately deriving a software system and its user interface from a common conceptual model can result in more accessible software tools. We demonstrate the feasibility of this approach by using it to build the user interface to a very general and highly automated system for generating data translation software. The major contribution of this work is:

- A demonstration of how accessibility problems can be greatly reduced by separating the design of a system from the design of its user interface, and by deriving both components directly from a conceptual model of the desired functionality. In the case of the Chameleon architecture, a conceptual model of the data translation problem is used to separately derive the Chameleon build phase and its user interface, the Translator Writer.

In applying this approach to the data translation problem, we make two additional contributions that are significant in their own right.

- **The Formal Characterization of Data Translation Problems:** The conceptual model of the data translation problem we present is the first formal, domain-independent characterization of this problem. This model is complete, in the sense that we can prove that it handles all the problems that may arise during data translation.

- **The Translator Writer:** The Translator Writer, a tool for specifying data translation software, was derived from this conceptual model in isolation from the rest of the Chameleon system. As we shall see in the next chapter, other data translation tools have been developed; however, the TW is the first tool for specifying high level data translations that is both accessible and domain-independent.
I.2 Organization of this Thesis

This thesis is organized as follows:

- Chapter II describes the process of using a conceptual model to separately drive the design of the user interface and the rest of the system, and introduces the data translation problem.

- Chapter III reviews the other work that has been done in this area.

- Chapter IV contains a formal characterization of the high level data translation problem, and discusses how the problems listed above are manifested in a specific domain, namely electronic document processing.

- Chapter V talks about the derivation of the user interface from the conceptual model, and describes the resulting language used to communicate with the architecture.

- Chapter VI describes the derivation of the Chameleon build phase from the conceptual model, and shows how the operations and entities in the conceptual model are implemented in the Chameleon architecture.

- Chapter VII describes the implementation of the Translator Writer, tells how the TW was used to build a set of translation routines for a non-trivial data translation, and what was learned from this experience.

- Chapter VIII lists the questions that still need to be addressed in this area, and directions for further work.
Chapter II

A Solution: Assuring Accessibility of Software Systems

Complex computer systems are frequently hard to use. It is therefore appropriate for computer scientists and software engineers to ask how complex systems can be made easier to use. In this chapter, we describe the accessibility problem in the context of the Chameleon project, and an approach to its solution that should be applicable to any large software system.

Most large software systems nowadays are built in stages. The first step is the preparation of a document that says, at a conceptual level, what the system is supposed to do. This document is usually known as a *functional specification*. This document is then used to guide the second step in the process, during which the system is designed in detail, and implemented. After a period of debugging and testing, the system is handed over to its users. This is not to suggest that these steps don't overlap, or get repeated during the life of the system, because in fact that is exactly what happens, and that is why the so-called waterfall model of software development has fallen into disfavor. However, as a coarse description of how real systems are actually built, these three steps seem to retain their validity.

There are three different models, or vocabularies, represented by each of these steps. When one begins to design a software system, one starts with an abstract model
that describes what the objects are that the system deals with, and what sorts of
relationships may exist between them. We refer to this model as the conceptual
model, and it is written in terms of a conceptual vocabulary. As the system described
in the functional specification is designed and implemented, the objects and operations
in the conceptual model are instantiated in terms of various procedures, data
structures, and so forth. During this process of design and implementation, these
procedures and data structures are described in terms of a system vocabulary. Finally,
there's the user, who has his or her own vocabulary, the user vocabulary, to describe
the objects and operations in the conceptual model. Basically, the user's vocabulary
represents his or her understanding of the conceptual model.

Given these three vocabularies, we can define accessibility as the extent to which
the functionality of the conceptual model is available to the user via the user
vocabulary. A system fails to be accessible when some part of the desired functionality is
- unavailable, perhaps by being mistakenly omitted from the functionality
  of the conceptual model, or
- available only through use (direct or indirect) of the system vocabulary.

Intuitively, this definition arises from the observation that we regard systems as hard
to use, or inaccessible, when we experience difficulty in doing what we want to do:
that is, when we experience difficulty in accessing the desired functionality.
II.1 The Traditional Approach to User Interface Design

The traditional approach to user interface design is shown in Figure 1. One starts with some sort of conceptual model of the various objects and operations to be manipulated and performed by the system. In a typical data processing environment, this model may be arrived at through discussion between the user and a systems analyst. In some cases, this conceptual model may be a formal mathematical description of the objects and operations. Using this conceptual model, someone writes a functional specification, typically in English, that describes what the application system is supposed to do and how it should work. This specification is then used to build the system. The user interface is designed along with the rest of the system. The functional specification may require that certain types of I/O devices be used, or that the user interface employ things like windows or menus. However, the user interface itself may not be designed until very late in the development of the system.

We have observed several problems that arise from this practice of making the design of the user interface just another part of the design of the whole system. One problem has to do with the desired functionality being omitted from the system, or perhaps it seems to be provided when in fact it is not. Such an interface tends to:

• Give incomplete or flawed access to the conceptual model.

Example: Decimal arithmetic, as implemented by most compilers. In most high-level programming languages (i.e. those without built-in extended precision arithmetic) the terminology and syntax are familiar, to a point; but the conceptual model of the real numbers in mathematics is not provided in its entirety, because of things like round-off errors and finite word lengths. Although it is impossible to capture the reals
Figure 1: The Traditional Approach
completely in a machine of finite size, it is possible to present the user with a model of the reals that is closer to the mathematical ideal by making sure that the user is protected from the ill effects of certain restrictions, such as word length, that are imposed by the hardware.

Another problem arises when the desired functionality is provided, but is accessible only through use of the system vocabulary. Such interfaces tend to:

- Use system-oriented terminology that is unfamiliar to the user.
  Example: In OS JCL, the command to access a file from a FORTRAN program might look like
  
  ```
  //SYSUT5 DD DSNAME=MHDSSYS.S99.L67,
  // DISP=OLD, VOL=SER=ABC123
  ```

- Require the user to adjust to a system-oriented language syntax.
  Example: In SPSS, the command used to do a factor analysis is
  ```
  FACTOR VARIABLES=varlist/TYPE=RAO/
  MINEIGN=.9/ROTATE=VARIMAX
  ```

Although the terminology might be natural to a statistician, the syntax probably would not be.

Finally, when user interface design is treated as just another part of the system development process, it tends to get postponed until very late in the project. User interfaces therefore suffer disproportionately from time and budget pressures.

If these problems arise whenever the traditional approach is followed, how is it that any decent user interfaces exist? This is a reasonable question, since this is the way we have written programs for many years, and some are quite accessible. In an intuitive sense, accessibility varies with the degree of overlap among the conceptual, system, and user vocabularies. In a system like Apple's MacDraw, for example, the
user's intuition, the system itself, and the conceptual model from classical geometry are very similar. For example, all three speak directly in terms of points, lines extending between them, and so forth. In other words, the conceptual model from geometry is already familiar to the user, and the implementation doesn't get in the way of the desired functionality. As a result, MacDraw is more accessible than most programs: the user finds it easy to access the desired functionality. However, systems become less accessible to the extent that the user is forced to use cryptic or unintuitive command sequences that bear a greater similarity to the system vocabulary than to the user or conceptual vocabularies. This phenomenon is familiar to users of more complicated systems, such as OS/MVS, or some expert systems that must devote great energy to keeping their users aware of where they are in the system and what they're doing [Carroll 87]. This is not to suggest that on-line help facilities, JCL command procedures, and other forms of user assistance should not be used in complex software systems; occasionally, they are necessary. It seems to us, however, that such mechanisms frequently come about as additions to systems that have already proven to be inaccessible to their end-users.

When the amount of overlap between conceptual, system, and user vocabularies becomes too small, then it will probably take several attempts to produce a satisfactory user interface. This is the point made by Gould and Lewis, and Draper and Lewis [Gould 85, Draper 85]. Given the rapid prototyping and executable specification techniques that have been developed over the last several years, it is possible for the user interface to be quite well specified at a relatively early point in the design process. The idea behind these techniques is that with each successive prototype, the user and the designer come closer to a mutual understanding of what the system should do and what its user interface should be. However, unless the prototype user
interface is derived from a conceptual model without contamination from the system, it may still fail to be accessible. The rapid prototyping technology then makes it easier to make whatever revisions are necessary to make the system accessible [Zave 86]. We agree with the proponents of rapid prototyping in believing that the design of the user interface merits early attention in the system design process. However, the state of the art in rapid prototyping does not, in our view, address the accessibility problem at its source: unless the system can be prevented from coming between the user and the desired functionality, inaccessibility is likely to be a problem.

II.2 Our Approach

We suggest a different approach to user interface design, one that emphasizes usability as a natural property of interfaces, as opposed to one that is achieved only with great effort. How can we build user interfaces and know that we have avoided the problem of inaccessibility? The key is to strive for interfaces that take advantage of the user's own (perhaps incomplete) understanding of the conceptual model, and avoid using system-oriented language or terminology in the interface. Both of these can be achieved by separating the design of the user interface from the design of the rest of the system, and by using the conceptual model to directly drive the design of the user interface, just as it directly (and separately) drives the design of the rest of the system (see Figure 2.)

The approach that we are suggesting consists of the following five steps:

1. Define a conceptual model of the desired functionality. This conceptual model is abstract, in the sense of being devoid of details relating to implementation and user interface. The specification of this conceptual model must identify the kinds of information required by each of the operations in order to provide the desired functionality.
Figure 2: Separating System and User Interface Design
2. Characterize the user in terms of a vocabulary that he or she can be expected to understand.

3. Derive the user interface from the conceptual model, while making sure that the resulting interface only uses terminology and syntax that is included in the user characterization.

4. Derive the system from the conceptual model. The system implementors are free to use any mechanism they wish in order to provide the desired functionality. (These last two steps may be done in either order, or in parallel, but they must precede the next step, hence the label "Design 1" in Figure 2.)

5. Connect the user interface to the system. Since the system and the user interface are both derived from the same conceptual model, in which the information requirements of each component are explicitly identified, it should be possible to make sure that all the information that the system needs is somehow provided by the user interface. (This is the second design step referred to in Figure 2.)

It is important for the conceptual model to be complete with respect to the desired functionality, since the system and the user interface are both derived from it. Similarly, the characterization of the user should be as accurate as possible, since otherwise we might either refer to concepts or vocabulary with which the user is not familiar, or oversimplify conceptual model concepts with which the user is familiar. We do need to assume that the user has some basic understanding of what is going on in the conceptual model, and that his or her vocabulary reflects this.
If this approach is used, the problems listed previously will then be much less likely to occur. Specifically,

• If the design of the user interface is directly inspired by the conceptual model, it is much more likely that the interface will give the user access to that model in a fashion that is both clear and complete;

• If the user interface and the rest of the system are designed and implemented separately, the chance of the user interface being contaminated by system terminology is greatly reduced;

• Similarly, if the user interface is derived in isolation from the rest of the system, it is unlikely to use a system-oriented syntax.

• Finally, the separation of system and user interface design means that the user interface can get attention early in the design process, thereby allowing time for measurements and minor adjustments.

We have just argued that the new approach is superior to the traditional approach because it solves the problems we observed with the traditional approach. However, one might wonder if our approach doesn’t introduce any new problems of its own. To see that this is not a cause for concern, we pause briefly to show that whenever the new approach might fail, the old approach will fail as well.

Consider each of the five steps individually: If we cannot identify some abstract or conceptual model of the desired functionality of the system, we are surely doomed no matter what approach we use. If there is no way to express this model’s ideas in terms of the user’s vocabulary because the user simply doesn’t have the necessary knowledge, he or she would also be unable to effectively use a system developed with
the traditional approach, contaminated (as it would be) with system-oriented detail. Once we have determined the user's vocabulary, then creating a user interface should be no more difficult than it is in the traditional approach. Similarly, creating a system that implements the desired functionality should present no extraordinary problems. Finally, if the system and user interface are derived from the same conceptual model, then causing them to fit together to form a cohesive system involves making sure that the user interface provides all the information that the system requires. Under the traditional approach, we would have to do this as well, but without the benefit of a conceptual model to serve as arbiter.

Foley has identified four layers that make up user interfaces: the conceptual layer, the semantic layer, the syntactic layer, and the lexical layer [Foley 83]. When we say that the conceptual model should be used to drive the design of the user interface, we mean that the conceptual model should serve as a foundation upon which the other user interface design decisions (i.e. those involving the other three layers) should be based. What we are suggesting is that the conceptual model and vocabulary can be used to drive the design of the user interface and the system in a way that is much less likely to introduce the problems of the traditional approach.

In order to demonstrate the feasibility of this approach, we need to use it in the construction of a real system. We have chosen to apply this approach in the design of the user interface to the build phase of the Chameleon architecture. As we mentioned in the previous chapter, the Chameleon architecture is used to support the specification, construction, and use of data translation tools. The rest of this chapter is devoted to a description of the data translation problem, and how the Chameleon architecture is used to solve it.
II.3 The High Level Data Translation Problem

There are many instances in computing where one can find several different software systems which seem to do similar things to similar sorts of data objects. Examples come to mind in several domains, including database management systems, statistical software packages, CAD/CAM systems, and text formatters. Certainly different software systems in a domain will vary to some degree in their functionality, yet users still ask questions like, "Why can't we use this data, which was entered using Software X, with Software Y? After all, X and Y do essentially the same thing."

As one gains experience in computing, one tends to answer such questions by appealing to the ideas of compatibility and interoperability, and especially the lack thereof. (The IEEE Standard Glossary of Software Engineering Terminology defines interoperability as the ability of two or more systems to exchange information and to mutually use the information that has been exchanged.) The expert replies, "Well, what you're asking would require us to translate object x, which Software X creates, into a corresponding object y, which could be used by Software Y. However, that is an expensive and time-consuming process."

This sort of dialogue has occurred often enough in large data processing installations which use a single machine architecture or operating system. The personal computer user also faces this problem, due to the wide variety of software packages available for applications such as spreadsheets, word processing, and database management. The proliferation of personal computers, and the increasing availability of communication media for connecting those computers to larger ones, means that heterogeneous computer networks are becoming common, and will become more common in the future. As a result, file transfer and electronic mail
facilities for moving data around such networks are coming into wider use. However, the lack of software interoperability limits the use that can be made of such data, and this lack prevents contemporary heterogeneous networks from achieving their potential as truly integrated computing environments. Hence, it is appropriate for computer scientists to examine this problem and to find a general solution for it.

We define a high level data translation as the process of translating a data object of some known structure into a semantically equivalent object for use by software other than that with which the original object was created. And, indeed, the process is in general quite expensive and time-consuming. The difficulties one faces when confronted with the task of planning and implementing a high level data translation project include:

- First, it is unlikely that any of the software people at the installation will view data translation as a specialty. On the contrary, it will probably be necessary to drag programmers away from their own work in order to write a number of one-shot, ad hoc data translation programs. This has at least two unpleasant consequences: These people will resent the distraction, and the software they were working on will be delayed.

- Second, because of the inconvenience involved, most computer installations do not do such data translations on a routine basis. Therefore, when the occasional imperative need does arise, it is unlikely that any investment will have been made in tools to make them easier.

This data translation problem has been noticed in particular domains, and several domain-specific efforts to help solve this problem have been made. The most work has been done in the area of database management, where the high level data
translation problem arises as one migrates data from one DBMS to another. Fry has surveyed this work, and concludes that general data base conversion tools are feasible [Fry 81]. The feasibility of high level languages to drive such tools has been demonstrated by IBM's XPRS project [Shu 77]. In the graphics area, a number of standards exist to facilitate exchange of graphical objects, but no software to support conversion of objects to and from these standards is widely available or accepted [Bono 85]. A similar situation exists in the domain of electronic document processing [Horak 85]. However, no domain-independent, general purpose data translation software support is yet available.

II.4 The Chameleon Translation Architecture

The Chameleon architecture is an integrated set of tools that support the specification, construction, and use of data translation software. The purpose of the Chameleon research project is to show that general-purpose, highly automated tools can be developed to support data translation, and that these tools can be used by non-specialists. The architecture is described in detail in Chapter VI and [Mamrak 87], so it is described here only in brief. The tools in the architecture belong to one of two phases:

- the use phase, in which translators are used and the user is presumed to have little or no computer background, and

- the build phase, in which translators are built and the user is a programmer with extensive knowledge of the application domain (e.g. spreadsheets, database, or graphics), but little or no knowledge of the internal details of the Chameleon architecture. We introduce the term Translation Engineer, or TE, to describe this person.
During the use phase, objects to be translated are run through a custom-built parser, which makes a first attempt at a translation. The user then has a chance to make changes or corrections to produce a final translation. During the build phase, the Translation Engineer specifies how the custom-built parser and user interface employed during the use phase are actually built. One important principle behind the operation of the parser is the idea of invertible attribute grammars [Yellin 87]. However, the notion of an invertible AG is both new and complex, and several build phase tools are involved in the construction and conversion of these invertible AGs into executable data translation programs. As a result, we can’t expect the Translation Engineer to have the time or inclination to learn about them. Instead, there is a build phase tool, called the Translator Writer (or TW) that serves as the interface between the Chameleon build phase and the Translation Engineer.

Several problems may be encountered during the specification of a data translation, and it is the responsibility of the Translation Engineer to communicate to the TW how these are to be handled. We will present a more careful description of these problems in Chapter IV, but in order to give the reader some feeling for what the problems are, we describe them here using the fragment of a Scribe™ document shown in Figure 3.

```plaintext
@device(x2700ii)
@begin(center)
@begin(bold)
Moby Dick
@end(bold)
Herman Melville
@end(center)
Call me Ishmael.
```

Figure 3: Fragment of a Scribe Document

Suppose that we wish to translate this document into a standard format that
emphasizes the semantics, or content, of the document, and is devoid of formatting or rendering information. The problems that need to be addressed are:

- **Specification.** The first problem is how to actually specify the mapping of a certain Scribe construct to its standard form equivalent. How should this be done?

- **Functionality Mismatch.** The second problem is what to do with those Scribe constructs that have no standard form equivalent, such as the "@device" command. Since this command (and many other Scribe commands) deals exclusively with formatting, it has no standard form equivalent. How should we deal with this?

- **Ambiguity.** The third problem is that there might be more than one way to interpret the Scribe document in terms of the standard form. The standard form may, for example, allow the document's author and title to be specified in either order, as long as both are indeed present. How is the translation software to know that Herman Melville wrote Moby Dick, and not vice versa?

- **Synonym Resolution.** The fourth problem is due to Scribe's ability to accept several different character strings as meaning the same thing. In the example, we could have said "@center(@b[...]...)", in place of the longer "@begin" constructs. How do we tell the translation software that these two are equivalent?

The Chameleon system has mechanisms to deal with these problems. However, these mechanisms involve relatively new and sophisticated concepts with which we cannot expect the Translation Engineer to be familiar. We must therefore provide a
user interface that gives access to this functionality, without forcing the Translation Engineer to learn a great deal about the internals of the Chameleon system. The first step in the process of making communication between the Chameleon architecture and the Translation Engineer feasible is to come up with a conceptual model of the data translation problem. This has been done: In the next chapter, we will describe this model and show that it is complete with respect to the desired functionality. This model will then be used to separately drive the design of the Translator Writer and the Chameleon build phase itself. These two components can then be brought together to form a cohesive system that provides access to the desired functionality.

II.5 An Example: The Chameleon Translator Writer

During the build phase, we will rely on the Translation Engineer to give us the information we need to specify the translation algorithms that map data objects into the standard form and back again. The processing logic in the build phase will take care of transforming this specification into an executable form.

In order to use our new approach in this context, we first need to characterize the conceptual model, the system vocabulary, and the user vocabulary. We have a conceptual model of the data translation problem, and we have used that model to determine the functionality of the various tools in the Chameleon build phase [Kaelbling 87]. In this model, the translation problem is described in terms of mappings, (specifically, substitutions, homomorphisms, and permutations) defined over certain sets of strings. We have identified the various situations that can arise in this model, and have proven that these are the only situations that need concern us (see Chapter IV).
This conceptual model was used in the design of the Chameleon architecture, including the build phase (see Chapter VI). However, as the system was built, we needed to use some ideas from computer science that would be unfamiliar to a Translation Engineer: Specifically, the Chameleon system vocabulary refers to invertible attribute grammars and reduce/reduce conflicts, to give just two examples. The system vocabulary therefore includes these terms, as well as the names of various data structures and procedures. As a result, were we not to take steps to prevent it, the build phase would be inaccessible to its intended users.

The user model is that which is possessed by the Translation Engineer. We assume that the Translation Engineer has some understanding of the abstract model, at least at an intuitive level. We also assume that he or she is an expert with what these concepts mean in terms of the local environment (see Chapter V). For example, if the Translation Engineer is using Chameleon to specify how some Scribe document is to be translated to some other format, we assume that he or she knows how the various types of objects in the abstract model (strings, tokens, sets of strings, and so forth) correspond with various objects in Scribe. While it is true that more complicated ways of characterizing user models have been suggested (e.g. [Jamar 86]), we feel that this is a reasonable way to characterize the user model for our purposes.

In order to show that this approach is feasible, we propose to use the conceptual model of translation that we have identified as the foundation for the design of the interface to the Chameleon build phase, and then to build the bridge between the user and system vocabulary. We will thus provide access to the conceptual model, via the system, by way of the user vocabulary. This will serve to demonstrate that our approach can be used to build a user interface to a software system of some complexity.
II.6 Summary

In this chapter, we have proposed a new approach to user interface design that promises improved usability of complex software systems, based on the use of a conceptual model to separately drive the design of a system and its user interface. We will demonstrate the feasibility of this approach in the domain of data translation. Specifically, we will

- construct a usable interface to a system for specifying high level data translations, and

- demonstrate a process that can be used to build user interfaces with high degrees of accessibility.
Chapter III

Related Work

The purpose of this chapter is to show how this work is related to other efforts, and, in particular, to show how this work constitutes an advance in the state-of-the-art. Our major contribution is in assuring the accessibility of complex systems by separating the design of the user interface from the rest of the system, and by driving both designs from a common conceptual model. In this chapter, we describe the previous work in the area of accessibility, paying special attention to those efforts that have advocated some degree of separation between the design of systems and their user interfaces.

III.1 User Interface Issues

Historically, the problem of accessibility has been relegated to the area of user interface design, where it has been lumped together with some other concepts under the vague heading of "user-friendliness." As Goodwin points out, when operating under tight schedules and limited budgets, software designers tend to put off the design of a system's user interface until late in the project, and then try to slap it together at the last minute. Part of this must be due to the eagerness of programmers to involve themselves with the internals of a system, and to make it as powerful, at least from their point of view, as the available resources will allow [Goodwin 87].
However, this tendency is also due to the fact that, until recently, nobody really knew what a good user interface was, or how to build one. As a result, the design of user interfaces has traditionally been much more of an art than a science. The objective scientific knowledge that is available has sprung largely from human factors work, of which Shneiderman's recent book is a very thorough and readable survey [Shneiderman 87].

As far as higher level user interface issues are concerned, no general method or theory for constructing usable software yet exists. We can, however, discern two distinct themes in the research done to date. The first theme concerns what might be called the iterative approach to user interface design: learn what you can from the user, build a prototype, take measurements as the user tries it out, and repeat the cycle until the user is satisfied [Hewett 86]. Each of these steps represent important and difficult problems in their own right, and much work has been done to solve them in a scientific, or at least more methodical, manner. The three subsections in this section are devoted to a summary of the work done within this theme. The second theme concerns the separation, or abstraction, of user interface issues in the context of software design. The work within this second theme is summarized in section III.2.

III.1.1 Learn from the User

The problem of characterizing users so that systems can be built to accommodate their needs dates from the time that computers first emerged from laboratories and began to be used for commercial applications. Although these user characterizations have been done for a long time, and their importance is widely recognized [Lundeberg 83, Rudawitz 83], it is still difficult to formally capture who the user of a system is, and what he or she knows. This is due to several factors:
• The difficulty users have with deciding what they want;
  
• The reluctance on the part of human experts to share their valuable knowledge, frequently called the knowledge engineering problem; and,

• The difficulty of organizing and representing such knowledge for use by computer systems [Kidd 85, Quinn 86].

The point here is that although no system can be designed without some characterization of who the user is and what he or she needs and knows, it is difficult to make such characterizations sufficiently formal. It would be nice, for example, to have a program that would read in a characterization of a user and produce a user interface that makes a given system accessible to that user. Unfortunately, no such program exists, nor is it likely to exist in the near future. However, system designers must have such characterizations to build their systems. When the developer's characterization of the user is inaccurate or incomplete, the resulting system runs the risk of being inaccessible. (In our case, the characterization of the user and his or her vocabulary is based on assumptions we make about the Translation Engineer's background and experience. This is discussed in detail in Chapter V.)

III.1.2 Build a Prototype

Brooks suggested in The Mythical Man-Month that system designers should "build one to throw away." This advice is especially appropriate to user interface work, since that is the part of a system that is most visible to the user and therefore has the highest impact on his or her perception and acceptance of the system. Building a prototype user interface, or several prototypes, allows the user to interact with the designers of the user interface, and the resulting interface is therefore much more
likely to meet with the user's approval, assuming that the basic system functionality is provided.

However, it is usually impractical and expensive to develop prototypes without some sort of software support tools. Several programming methodologies have been developed with the goal of shortening the period of time between initial specification of a system and the appearance of a prototype. For example, much has been written in the last several years about rapid prototyping and the use of executable specification languages (ESLs). The most famous executable specification language is probably Zave's PAISLey system [Zave 86], which has proven useful in the construction of real-time communication software. The idea behind executable specification languages is to produce a working system from an incomplete specification at minimal cost. The system may indeed "work" only in a very qualified sense, since large portions of the system's functionality may not only be unimplemented but in fact only just barely specified at the time the prototype is built. ESLs are quite well suited to the iterative design of user interfaces, since a specification of a user interface may be worked out in great detail, and thoroughly tested, before any of the other system components have been fully designed.

Other software development techniques, aside from ESLs, can be used to develop prototype interfaces. Wasserman, for example, has shown that state transition diagrams can be used to specify user interfaces, and that since such diagrams are easy to create and modify in comparison to the equivalent code written in a normal programming language, this is a very reasonable way to build prototype interfaces [Wasserman 86]. Easy production of interface software is also one of the motivations behind graphical programming, since these techniques allow the designer to see the results of his or her decisions immediately [Newman 68, Haraw 80].
III.1.3 Take Measurements

Since usability of a system depends not only on the system but its inherently unpredictable human users, completely objective and reliable measures of usability are hard to come by. Most of the results in usability have come about through controlled experiments with human subjects. As Shneiderman points out, it is difficult to measure more subtle effects when human subjects vary so greatly in their innate programming ability. However, several workers have attempted to measure usability using static, easily measured properties of the interface software. Lindquist has shown that software science can be used to measure the complexity of user-computer dialogue systems [Lindquist 85]. His approach is to model a user interface with pseudo-code, and then count the number of operators and operands. These counts are then used to compute software science metrics such as effort and level (Halstead’s $E$ and $\lambda$, respectively [Halstead 77].) Lindquist’s point is that it is reasonable to suspect that interfaces with excessively high values for these metrics are inaccessible, although exactly what values are excessive would vary with both the user’s background and the system’s functionality. To put this another way, it might be quite reasonable to ask one class of users to use an interface with certain values of these metrics, yet it might be just as unreasonable to ask another class of user to use the same interface to access the same system. Lindquist’s approach would therefore seem to be most useful when a user characterization is available that lets us deduce ranges of acceptable and unacceptable values for these metrics. Reisner [Reisner 81] and Card, Moran and Newell [Card 83] model user-computer dialogue with formal grammars, and use properties of these grammars to predict the usability of the system being modeled and suggest ways to improve it. These approaches suffer from the same shortcomings as Lindquist’s, namely that the results of these measurements need to be carefully interpreted in light of the user characterization. Although it may be
difficult to say with confidence that a given interface is accessible and another is not, basing such a judgment solely on the values of these (or any other) metrics, these metrics may be used to say that a given interface somehow corresponds more closely to a user's mental model. This is an important question, since such a user mental model would be a very useful part of the user characterization, yet it is sometimes difficult for users to describe their mental models. One such study was done by MacDonald, who used user interface measurements to figure out what sort of mental models the users were developing of a system that they were learning how to use [MacDonald 86].

III.2 Toward Separation of System and User Interface

Our approach of separate design of system and user interface promises to make this iterative process shorter and less costly. Other people have suggested that certain aspects of user interface processing be separated from the rest of the system. Parnas's work on modularization hinted at this idea fifteen years ago, since specialized knowledge of the form and operation of a user interface is information that might wisely be hidden from the rest of the system [Parnas 72]. More recently, the idea of separating the design of the user interface from the rest of the system has been put forward by Yunten and Hartson [Yunten 86]. They advocate the use of two specialists, a human factors expert and a software engineer, to separately develop the dialogue component and the rest of a system. However, they do not separate the two completely, since the human factors expert and the software engineer are expected to work together to develop the main control structure that drives the entire system. Although we of course agree that the user interface should be designed apart from the rest of the system, it seems that this approach has two problems. First, there is no explicit provision for a common view of the problem, from which the two experts are
supposed to take guidance, such as that provided by an established conceptual model. Second, the cooperation of the two experts in producing the main control structure invites contamination between the two components. We feel that the two components can be separated more completely.

Once the design of the user interface is separated from the design of the rest of the system, it becomes reasonable to contemplate ways of producing user interface code automatically. For example, the development of window management systems for personal computers and workstations [Scheifler 86], systems that automatically construct dialogue managers [Hix 86], and sophisticated user interface management systems [Singh 86] not only make it easier to produce user interface code, but also allows the design of the user interface to be separated from the design of the rest of the system as far as coding of individual modules is concerned.

III.3 Critique of Previous Work

In this chapter we have described the work done to date within two themes of user interface research: the iterative approach, and the separation of user interface and system design issues. What makes our work different from what's been done before? We can draw three conclusions:

• It is apparent that user interface design is still a hard problem, and that it will remain hard for some time to come. The problem is made harder by trying to design the user interface along with, and as a part of, the rest of the system. We have not seen any work that advocated the development of a conceptual model and subsequent derivation of separate user interface and system components.
• Foley has noted that in a software system, the user interaction takes place on several levels: conceptual, semantic, syntactic, and lexical [Foley 83]. Much user interface research has focused on syntactic and lexical issues. As a result of the central role in which we place the conceptual model, our work directly addresses the accessibility issue at the conceptual and semantic levels.

• The techniques of rapid prototyping and measurement retain their usefulness under our approach. In fact, the use of a conceptual model to drive the design of a user interface, as well as the separation of user interface and system design, will make it easier to develop a high quality interface in a reasonable amount of time.

In the next chapter, we introduce a conceptual model of data translation that will be used in the derivation of an accessible data translation system.
Chapter IV

Data Translation and Related Problems

In this chapter, we present a formal model of data translation. The principal reason for doing this is to establish a solid foundation for the Chameleon approach to data translation in general. In particular, as we shall see in the next two chapters, this model will be used in the design of the Chameleon build phase and of the build phase’s user interface. We also intend to show how the four problems given as examples in Chapter II are explained with this model.

IV.1 A Formal Model of Data Translation

We have defined "data translation" as a process of converting objects belonging to one software package into a format in which they can be used by other packages. In this section, we describe a formal model of this process.

IV.1.1 The Standard Form Approach to Data Translation

Suppose that we are dealing with a domain containing \( n \) sets of data objects, and that we are interested in translating certain objects in one set, say \( X_i \), into equivalent objects in each of one or more other sets \( X_{j}, X_{k}, \ldots, X_{n} \). One way to do this would be to build individual translators, each mapping \( X_i \) to \( X_j \) for \( i \) and \( j \) between 1 and \( n \). This will be called the pairwise approach. An alternative is to first define a standard form,
called $S$. Then, for each of the $X_i$'s ($1 \leq i \leq n$), we construct two translators: one mapping $X_i$ up to $S$, and the other mapping $S$ down to $X_i$. Then a translator mapping objects in $X_i$ to $X_j$ can be built by composing the translator mapping $X_i$ to $S$ with the translator mapping $S$ to $X_j$.

In deciding whether to use the pairwise or standard form approach, it should be noted that as the number of sets in the domain increases, the standard form approach enjoys a significant advantage in complexity. With the pairwise approach, as $n$ increases, the number of translators needed increases at a rate proportional to $n^2$. Using the standard form approach, the number of translators needed is $2n$. The standard form approach reduces the general data translation problem to the problem of building translators between the standard form and each of the $X_i$'s. The $X_i$'s are called nonstandard forms. Translation from the standard form down to a nonstandard form will be called translation down, and translation from a nonstandard form up to the standard form will be called translation up (see Figure 4.)

IV.1.2 Standard Forms as Braced Languages

To discuss the problems of translation in a formal way, we need to make the notion of standard form more precise. In a very basic sense, standard forms are languages used to express the information contained in data objects. Kaelbling has discovered that it is reasonable to describe standard forms as instances of a certain subset of the context-free languages, which he has named the Braced languages [Kaelbling 87].
Figure 4: The Standard Form Approach
Definition 1: A Braced language is the intersection of a context-free language and a regular set generated by a grammar $B_k=(V, T, P, S)$, where

- $V$ is the set of nonterminals \{S, B, C\}
- $T$ is the set of terminals \{a_1, \ldots, a_k, z_1, \ldots, z_k, c_1, \ldots, c_m\}. The $a_i$'s are opening braces, the $z_i$'s are closing braces, and the $c_i$'s are nonbrace characters
- $P$ is the set of productions containing only
  - $S \rightarrow a_i B z_i$
  - $S \rightarrow a_i C z_i$
  - $B \rightarrow a_i B z_i$
  - $B \rightarrow a_i B z_i B$
  - $B \rightarrow a_i C z_i$
  - $B \rightarrow a_i C z_i B$
  - $C \rightarrow c_j C$
  - $C \rightarrow \epsilon$
- and $S$ is the start symbol.

$L(B_k)$, the language generated by $B_k$, can be thought of as the balanced strings containing $k$ types of parentheses that brace strings over the alphabet of $c_i$'s. It is reasonable to use Braced languages as standard forms because each of the different brace symbols can be used to explicitly represent some information. For example, we can define a standard form for simple documents as follows:
The $c_j$'s are the standard alphabet and punctuation marks. If we use the symbol $c$ to denote the set of strings of any length containing the nonbrace symbols, 

$$(c_1c_2^* + \ldots + c_m)^*$$

then the regular expression for this standard form is

$$a_1 a_2 c_2 a_3 c_3 a_4 c_4 z_1$$

This regular expression causes strings in this standard form to resemble:

```latex
<document>
<title> ... <end title>
<author> ... <end author>
<body> ... <end body>
<end document>
```

For example, the standard form object

```latex
<document>
<title>
Moby Dick
<end title>
<author>
Herman Melville
<end author>
<body>
Call me Ishmael.
<end body>
<end document>
```

would be equivalent to the Scribe document shown in Figure 3.

The most desirable property that a standard form may possess is expressive power, meaning that it is capable of expressing all the information found in the (class of) data objects to be translated. It turns out to be helpful if the strings in this language are easy to parse, and if the information in them is explicit; but none of these properties are strictly necessary.
IV.1.3 Translation Functions

If we view standard forms as formal languages, then the purpose of the translation functions is to map the members of these standard form languages, which we will call standard form strings, to and from the nonstandard form strings that express the same information.

We need to formally define the mappings that are used for translation up to the standard form and down to a nonstandard form. We begin with the following definition from [Hopcroft 79].

Definition 2: Let $A$ and $B$ be finite alphabets. Then a substitution $s$ is a mapping from $A$ onto subsets of $B^*$. The mapping $s$ is extended to strings in $A^*$ as follows:

- $s(\epsilon) = \epsilon$,
- $s(xy) = s(x)s(y)$ where $x \in A$ and $y \in A^*$.

The mapping $s$ is extended to languages by defining $s(L) = \cup \{s(x) \mid \forall x, x \in L\}$.

For example, let $s(0) = a$ and $s(1) = b^*$. Then $s(010)$ is the regular set $ab^*a$. If the goal is translation, presumably the substitution would map each string in $A^*$ to a set of strings in $B^*$ that mean the same thing. If each of these sets of strings in $B^*$ is singleton, the substitution is a called a homomorphism.

Definition 3: A homomorphism $h$ is a substitution such that $h(a)$ contains a single string for each $a$ in $A$.

We abuse the notation slightly by taking $h(a)$ to be the string itself, as opposed to the set containing the string. For example, if $h(0) = \{x\}$, $h(1) = \{yy\}$, and $h(2) = \{x\}$.
then we say $h(012) = xyyx$. Kaelbling has shown that every context-free language can be expressed as the homomorphism of a Braced language. As a result of this, we can use Braced language notation and terminology in reference to any translations between nonstandard forms expressible as context-free languages.

**Definition 4:** The inverse substitutional image of a string $w$ is the set of strings in $A^*$ that map to $w$:

$$\mathcal{S}^{-1}(w) = \{x \mid w \in s(x)\},$$

and for a language $L$,

$$\mathcal{S}^{-1}(L) = \{y \mid s(y) \cap L \neq \emptyset\}.$$

The mappings that we have defined so far are order preserving, in the sense that information appears in the nonstandard form object in the same order in which it appeared in the standard form object. In practice, this is too restrictive. During translation down, for example, it may be necessary to change the order in which translated substrings appear in the resulting nonstandard form object. To account for this, we introduce the notion of a permutation function.

**Definition 5:** Let $h$ be a homomorphism and $x$ a string belonging to a standard form Braced language $B_k$, and let

$$h(x) = h(x_1) \ldots h(x_n)$$

for some $n$, where each $x_i$ is a member of $T$, the set of terminals in $B_k$. (That is, each $x_i$ is an opening brace, a closing brace, or a nonbrace character.) Then a permutation $p$ is a function, mapping $h(x)$ to the nonstandard form, defined as follows:

$$p(h(x)) = h(x_{i_1}) \ldots h(x_{i_n})$$

where $1 \leq i_j \leq n$, and for $j \neq k$, $i_j \neq i_k$. 
Note that, by this definition, permutation functions are one-to-one and therefore have inverses. We can now define translation down and translation up.

**Definition 6:** Let \( B_k \) be a Braced language standard form, \( x \) be an arbitrary element of \( B_k \), \( h \) be a homomorphism, and \( p \) be a permutation. Then translation down is a function, mapping the standard form to the nonstandard form, defined as follows:

\[
t_d(x) = p(h(x))
\]

Since \( p \) is one-to-one, and therefore \( p^{-1} \) is a function, then translation up is a function, mapping the nonstandard form to the standard form, defined as follows:

\[
t_u(y) = s^{-1}(p^{-1}(y))
\]

where \( s^{-1} \) is the inverse substitutional image defined by \( s \), and \( y \) is an arbitrary nonstandard form string.

Figure 5 shows how the function \( h \) and \( p \) are composed to form translation down. Suppose \( x \) is a standard form string, where \( x = x_1 \ldots x_n \). Each \( x_i \) is mapped by the substitution \( s \) to a set of corresponding strings. For each \( i \), exactly one member of \( s(x_i) \) is selected to serve as \( h(x_i) \), thereby defining \( h(x) \). All these \( h(x_i) \)'s are then rearranged by \( p \) and concatenated to form a valid nonstandard form object \( y \).

Figure 6 shows how the functions \( p^{-1} \) and \( s^{-1} \) are composed to form translation up. Let \( y \) be a nonstandard form object in the range of \( p \). Then \( (y) = y_1 \ldots y_n \), and \( p^{-1}(y) = y_1 \ldots y_n \), where each \( y_i \) is an element of \( s(x_i) \) for some \( x_i \) with \( 1 \leq i \leq n \). (Note that \( y_i \) may or may not be equal to \( h(x_i) \) for a given \( x_i \).) Applying \( s^{-1} \) to each \( y_i \) gives \( x_i \), and the resulting string \( x_1 \ldots x_n \), comprises a standard form object \( x \).
$h(x)$ is a member of the set $s(x)$

Figure 5: Translation Down as $p(h(x))$
maps any string in $s(x)$ back to $x$

undoes the re-arrangement

Figure 6: Translation Up as $s^{-1}(p^{-1}(y))$
IV.2 Data Translation Problems

In this section we demonstrate that the set of four problems mentioned in Chapter II, specification, functionality mismatch, synonym resolution, and ambiguity, are the only problems that can arise during data translation, given the formal model of data translation we have just established. Considering the first of these problems: How should we specify the translation up and down functions? Given a standard form $SF$, we first specify the substitutions and permutation functions that map substrings that comprise a standard form object to the equivalent strings in $NSF$. We can then define the homomorphism component of translation down by choosing one string from the each of the sets specified by the substitutions. Continuing with the Scribe example from earlier in this chapter, we can define a sample translation down substitution, which happens to be a homomorphism since each of the sets of nonstandard form strings is singleton, as follows:

$h("<document>")=\{"@device(x2700ii)"\}$

$h("<title>")=\{"@begin(center)\n@begin(bold)"\}$

$h("<author>")=\{"\}$

$h("<body>")=\{"\}$

$h("<enddocument>"=\{"\}$

$h("<endtitle>")=\{"@end(bold)"\}$

$h("<endauthor>")=\{"@end(center)"\}$

$h("<endbody>")=\{"\}$

$h(c_i)=\{c_j\}, \forall i, 1 \leq i \leq m$

Since the order in which information appears is the same in the standard form and nonstandard objects, we can complete the definition of the translation down function by taking $p$ to be the identity function. We will have to rely on the Translation Engineer to specify these translation functions. If we can do this, we will have solved the specification problem.
We now show how the three remaining data translation problems of ambiguity, functionality mismatch, or synonym resolution, may arise.

**Theorem 7:** Given a standard form $SF$ consisting of a Braced language, and the specification of a substitution $s$, a homomorphism $h$, and a permutation function $p$ that map strings in $SF$ to strings in a nonstandard form $NSF$, the only data translation problems that may arise are functionality mismatch, ambiguity, and synonym resolution.

This result follows directly from the following lemmas.

**Lemma 8:** Given $SF$, $NSF$, $s$, $h$, and $p$ as above, the only data translation problems that may arise during translation down are functionality mismatch and synonym resolution.

**Proof:** Consider Figure 7. Suppose that $s$ maps a given standard form string $x$ to a set of nonstandard form substrings. Then either $s(x)$ is empty, contains exactly one nonstandard form string $y$, or contains two or more nonstandard form strings $y_1, y_2$, etc. We consider each case individually:

1. **If $s(x)$ is empty.** If $s(x)$ fails to map to any string in $NSF$, then $x$ must contain some information that cannot be expressed in the nonstandard form, and is therefore lost during translation down. This form of information loss is due to **functionality mismatch**.

2. **If $s(x)$ contains exactly one nonstandard form string, $y$.** If $s(x)$ contains exactly one string, then there’s no problem: we can define the homomorphism component of translation down $h$ so that, for this particular $x$, $h(x) = s(x)$.

3. **If $s(x)$ contains two or more strings.** We must choose a single string in $s(x)$ to serve as the value of $h(x)$, and make sure that
Figure 7: Three Possibilities for $s(x)$
translation up recognizes and maps all the strings in $s(x)$ back to $x$.

This is the synonym resolution problem.

**Lemma 9:** Given $SF,$ $NSF,$ $s,$ $h$ and $p$ as above, the only data translation problems that may arise during translation up are functionality mismatch and ambiguity.

**Proof:** Consider $s^{-1}$, the set of standard form strings in the domain of $s$ (see Figure 8.) Suppose $y$ is a nonstandard form string. Since $p$ is one-to-one, $p^{-1}(y)$ is a nonstandard form substring in the range of $s$. Then the set of standard form strings mapped to $p^{-1}(y)$ by $s$, $s^{-1}(p^{-1}(y))$, is either empty, contains exactly one standard form string, or contains more than one standard form string. Again considering each case individually,

1. If $s^{-1}(p^{-1}(y))$ is empty. This can only occur if the standard form is not capable of expressing the information contained in $y$. In this case, the standard form has insufficient expressive power, and this represents functionality mismatch.

2. If $s^{-1}(p^{-1}(y))$ has exactly one member. Having $s^{-1}(p^{-1}(y))$ possess exactly one member poses no problem: we can define the value of the translation up function when applied to $y$ as $s^{-1}(p^{-1}(y))$ without difficulty.

3. If $s^{-1}(p^{-1}(y))$ has more than one member. If $s^{-1}(p^{-1}(y))$ contains more than one member, then there must be (at least) two standard form strings that are mapped down to $y$ by translation down. If we assume that these standard form strings express different meaning, then we are left in doubt as to how translation up should be defined when applied to $y$. We refer to this as the ambiguity problem.
Figure 8: Three Possibilities for $s^{-1}(x)$

- $s^{-1}(x)$ is empty (functionality mismatch.)
- $s^{-1}(x)$ is not empty
  - $s^{-1}(x)$ has exactly one element, ok.
  - $s^{-1}(x)$ has more than one element (ambiguity)
We have assumed that the standard form and nonstandard form use a common character set, except for the brace characters. To see that relaxing this assumption does not change the formal model, suppose that we have a standard form that uses a certain set of characters \( c_1, c_2, \ldots, c_m \), and a nonstandard form that uses a distinct set of characters \( d_1, d_2, \ldots, d_n \). If some standard form character \( c_k \) has a special meaning in the nonstandard form, and does not appear as a character in its own right, we incorporate the appropriate mapping in the definition of the translation down homomorphism. This problem occurs, for example, in translating from a standard form down to LaTeX. The backslash character, "\", is used in front of keywords and commands, and does not appear by itself in running text. When the user really wants a backslash, two of them are typed in together. If the standard form character set includes the backslash, translation down must make sure that backslashes in the text are doubled in the output LaTeX document.

In this chapter, we have described a formal model of data translation. Using this model, we have shown that when considering data translation using standard forms that are defined as Braced languages and translation functions defined in terms of substitutions, homomorphisms, and permutations over these languages, the only problems that can arise are specification, ambiguity, synonym resolution, and functionality mismatch. Unfortunately, the model does not tell us how to solve these problems; it doesn’t say how the translation functions should be specified, or how they are best constructed. These are matters that need to be addressed during the design of the system and its user interface. In the next chapter, we begin to address these questions.
Chapter V
The User Conceptual Framework

This chapter describes the derivation of the user interface for the Chameleon build phase. The approach is novel, in that the user interface is being designed separately from the design of the rest of the Chameleon system. In particular, both the build phase itself, and the user interface to the build phase, known as the Translator Writer (or TW), are both derived from the conceptual model described in Chapter IV.

This chapter has two purposes. First, we describe the process used to derive a user interface from a conceptual model, in terms of specific steps and justifications for them. Second, we describe the various individual decisions made in this particular application of the methodology, namely the design of the Translator Writer.

As we mentioned in Chapter II, Foley and VanDam have suggested that user interface languages consist of four separate levels: the conceptual, semantic, syntactic, and lexical [Foley 83]. Foley and VanDam regard the conceptual level as the set of key application concepts that must be mastered by the user. In the semantic level, the exact functionality of the entities in the conceptual model and the effects of the various operations that can be applied to them are defined. The syntactic level defines the sequence of inputs and outputs, and the lexical level describes the actual input/output tokens.
In our approach, since the conceptual and user models are distinct, this layered model of user interfaces must be modified. The conceptual model, from which we derive the user and system models, is indeed made up of key application concepts. However, we cannot agree with the idea that these concepts must be mastered by the user. In particular, we do not expect the user to master the mathematical arguments involved in the conceptual model.

The heart of the problem is the relationship between the (separated) user and conceptual models and the semantic level. Foley and VanDam say that this level is where the exact functionality of the conceptual model is defined. In our approach, the exact functionality of the conceptual model has two manifestations: one is the user interface, and the other is the system. Both of these are derived from the conceptual model, and draw their validity from it. In designing the user interface, the semantic level is the interpretation of the conceptual level’s functionality in terms of the user model. (Similarly, the system model will be an interpretation of the conceptual level’s functionality, written in a vocabulary appropriate for that purpose.) Ideally, this interpretation would completely express the functionality of the conceptual model, using only those terms we know to be in the user vocabulary.

However, this raises another question: How close do we have to come to this ideal? Or, in other words, are there situations in which an approximation of the meaning of a given item in the conceptual model may be used in the user interface? If the approximation we choose is not entirely within the user’s vocabulary, then we risk making some aspect of the functionality inaccessible, or perhaps we incur some cost by making the user do some learning before he or she can use the system effectively. On the other hand, we might err on the side of over-simplification. This has at least
two potential drawbacks: We might gloss over some non-trivial aspect of the functionality in the conceptual model, thereby preventing complete access to that functionality; We might also, through over-simplification, lead the user to believe that he or she may perform some sequence of operations that is in fact invalid with respect to the conceptual model. This might not be easy to detect within the (over-simplified) user interface, but should be detected on the system side if the code is written to check the validity of input requests.

As we design the syntactic level of the interface, a language that we are calling MDL (for Mapping Description Language), we take care to stipulate the meaning each construct has with respect to the user's interpretation of the conceptual model. The lexical level determines how the tokens, or terminal symbols, of this language are formed. In traditional programming languages, the lexical level design is incorporated into a lexical analyzer, which produces a stream of tokens for the parser. In an environment with a graphics orientation, such as the one contemplated by Foley and VanDam, the lexical level is more complicated, since there may be several types of I/O devices involved. We do not at the moment expect MDL to have any special needs at the lexical level.

V.1 Deriving the User Interface

The approach we will follow contains four steps:

1. List the assumptions that we make about the user, which we use to decide what is and is not in the user's vocabulary.

2. List the objects and operations in the conceptual model and the corresponding terms or constructs in the user vocabulary. Some of these
correspondences may be imperfect; indeed, that may happen frequently in a non-trivial system. There may also be a number of ways to represent the same concept in the user vocabulary. If so, then we may wish to design the user interface so that any one of these terms may be used to access the desired functionality. (It has been shown that providing a number of ways to access a given function can lead to dramatic improvements in accessibility at the lexical and syntactic levels, since users will then be free to access the functionality in whatever way they feel comfortable [Furnas 87].)

3. Describe the syntax of a language, the semantics of which are defined in terms of the user model.

4. After this language is defined, consider issues of lexical analysis and presentation.

The remainder of this chapter is organized as follows: First, we describe the user, and in particular, the user’s vocabulary. We then present a list of the entities in the conceptual model along with the corresponding items from the user’s vocabulary. We then show how each of the six tasks that comprise translation in the conceptual model (described in Chapter IV) are manifested in the user’s vocabulary. We then present a sample MDL specification, and explain each construct and its meaning (or semantics) with respect to the user model.
V.2 The User Model and Vocabulary

We have made certain assumptions about the user of the TW, an individual we refer to as the Translation Engineer. We explain our list of terms in the user vocabulary by appealing to these assumptions. These assumptions are:

1. The TE is an expert in the nonstandard form.

2. The TE is familiar with BNF or a similar notation, and is therefore at least casually acquainted with the basic ideas of formal grammars, including productions, and the difference between terminal and non-terminal symbols.

3. The TE is sufficiently familiar with the standard form so that he or she can read an arbitrary standard form object and tell what it means, if anything, in the nonstandard form.

4. The TE understands some basic concepts of programming, including the idea of expressions, variables, and character strings.

Based on these assumptions, we can assume that the user's vocabulary includes the following terms:

- grammar
- BNF
- production
- nonterminal symbol
- terminal symbol
- start tag
- end tag
- string
V.3 The Conceptual Model

The conceptual model presented in Chapter IV describes several entities and operations. Some of the vocabulary is rather specialized or mathematically oriented. We are particularly interested in how the problems of functionality mismatch, ambiguity, and synonym resolution can be related to the user. In order to assure ourselves that everything in the conceptual model can be handled in the user model, we present here a list of the conceptual model entities and the six situations given in Chapter IV, and their interpretations with respect to the user model. In this listing, the interpretations themselves are written using the user vocabulary. The interpretation is taken from the Translation Engineer's point of view. For each entity in the conceptual model, we give a description of that entity in terms of the Translation Engineer's vocabulary. Each of the six tasks is described in terms of how the situation is presented to the TE, and how he or she is expected to respond. In this way, we show that the entities and tasks in the conceptual model have meaning to the Translation Engineer, and that therefore the entire functionality of the conceptual model is accessible to him. Further, it is important to note that this description is not contaminated with system-oriented concepts or terminology, since these concepts and terminology are developed separately.
standard form  A language that is used to capture the meaning of a class of data objects so that those objects may be shared among different software packages. Certain strings, known as start and end tags, are used to make the meaning of the data object explicit. The standard form is defined with a BNF grammar that describes the various start and end tags and the order in which they may appear.

nonstandard form

A language used to describe a set of data objects that are considered legal by a certain software system. The TE is assumed to be an expert in the nonstandard form, in the sense that he or she could look at an arbitrary standard form object and produce an equivalent nonstandard form object if one existed. However, we cannot assume that the TE thinks of the nonstandard form as a formal language, simply because he or she might not have a formal grammar for the nonstandard form.

translation up  A mapping from nonstandard form strings to standard form strings.

translation down

A mapping from standard form strings to nonstandard form strings.

start tag  A terminal symbol in the standard form that provides some information pertaining to the meaning of the text delimited by the start tag and the corresponding end tag that follows it.

end tag  A terminal symbol in the standard form that marks the end of a portion of text affected by the corresponding start tag.
non-brace string

Text in a standard form document with no embedded start or end tags.

substitution

A mapping that associates a string with a set of strings, where the set may have zero or more elements.

homomorphism

A mapping that associates a string with a set of strings, where the set has exactly one element.

permutation

A mapping that, given a string composed of a set of substrings, rearranges those component substrings to form another string. In the new string, the substrings may appear in any order, but each must appear exactly once. In particular, translation down may be thought of as the composition of two functions: a homomorphism that associates each string in the standard form with a single string in the nonstandard form, and a permutation that re-arranges a number of such nonstandard form strings to form a valid nonstandard form object.

inverse substitution

A mapping that associates a given string, say x, with a set of strings Y such that each of the members of Y is mapped to x by a certain substitution. In particular, translation up may be thought of as an inverse substitution that associates each nonstandard form string with a set of standard form strings such that those standard form strings are mapped to the given nonstandard form string by a translation down substitution.
We can now show how each of the six tasks in the formal model are described to the Translation Engineer, and how he or she may be expected to respond.

**TD: functionality mismatch**

A standard form string, or set of such strings, that is mapped to no string in the nonstandard form. The TE is asked to decide how such strings are to be handled, i.e. should they be ignored, cause an error message, or be passed through to the nonstandard form object as is.

**TD: homomorphism**

A standard form string that may be mapped to exactly one nonstandard form string.

**TD: synonym resolution**

A standard form string that is mapped to any one of a set of several distinct nonstandard form strings. The TE must indicate which one of these nonstandard form strings is to be produced during translation down, but all the strings in the set of nonstandard form strings should be regarded as having the same meaning, and therefore should be mapped to the same standard form string during translation up.

**TU: functionality mismatch**

A nonstandard form string that has no meaning in terms of the standard form. The TE must specify how such things are to be handled, e.g. ignore them, raise an error condition, or mark them distinctly and pass them through.
TU: inverse substitution

A nonstandard form string that is mapped to exactly one standard form string. Since the inverse substitution case is handled automatically by the system, we require no input at all from the TE in this case.

TU: ambiguity

A nonstandard form string that is mapped to one of several distinct standard form strings. The TE may indicate that certain nonstandard form strings are to be mapped to specific standard form strings, or that the decision is to be made by a person during the use phase.

We have just shown that each of the entities and tasks in the conceptual model of data translation can be presented to the Translation Engineer in a way that gives the TE access to all the functionality of the conceptual model, without running the risk of system contamination. We can now present the language that the TE uses to access this functionality.

V.4 An MDL Specification

In this section, we present a sample MDL specification and a description of how the various MDL constructs give the TE access to the functionality in the conceptual model. A BNF description of MDL is given in Appendix A. The TE does not write an MDL specification from scratch. Instead, as we will show in Chapter VI and Chapter VII, the starting point is a BNF description of the standard form. The TE decorates the BNF with MDL constructs to describe the translation. (Figure 10 shows the decorated BNF for a standard form.)
We now describe the various MDL constructs shown in Figures 9, 10 and 11, explaining how each gives the user access to some aspect of the functionality of the conceptual model. An MDL specification has three parts: A Declarations section, a Standard Form section, and a Nonstandard Form section. Figures 9, 10 and 11 constitute a complete MDL specification. The line numbers were added for the sake of this explanation, and they are ignored by the Translator Writer.

V.4.1 MDL Declarations

In the first section of an MDL specification, the Translation Engineer provides information about several aspects of the translation, including the character sets used by the standard and nonstandard forms and the definition of synonym sets.

DEclarations

1  KEYWORD "@"
2  STANDARD FORM "&LT;" IS "<"
3  NONSTANDARD FORM "@" IS "@"
4  NONSTANDARD FORM "FROM" IS SPECIAL IN math
5  CLASS device_spec "@device(lpt)" | "@device(postscript)" IS "@device(lpt)"
6  CLASS doc_type "@make([a-zA-Z0-9]+)"
7  CLASS style_stuff IS IGNORED
8  CLASS font_stuff "@font([a-zA-Z0-9]+)"
9  IS IGNORED
10  CLASS value_stuff "@value([a-zA-Z]+)"
11  IS BRACKETED

Figure 9: An MDL Declaration Section

The KEYWORD command (1) indicates that strings that have some special meaning in the nonstandard form begin with a certain sequence of characters. If the TE uses a nonstandard form string in the specification
of a translation, and this string does not begin with the designated character sequence, then that string might be confused with ordinary data during translation up, and the TW issues a warning message to this effect.

STANDARD FORM <characters> IS <characters>

The STANDARD FORM command (2) is used when the standard and nonstandard forms use different character sets. The STANDARD FORM command indicates that when the first character sequence is encountered in the standard form document, it is to be replaced with the second character sequence as part of translation down.

NONSTANDARD FORM <characters> IS <characters>

The NONSTANDARD FORM command (3) is the counterpart to the STANDARD FORM command form. This command specifies that occurrences of the first character sequences are encountered in a nonstandard form document during translation up, they are to be replaced with the second character sequence.

NONSTANDARD <characters> IS SPECIAL IN <context>

This form of the NONSTANDARD command (4) indicates that a certain nonstandard form string is to have special significance, but only in a certain context. (See the &BEGIN and &END commands, below.)

CLASS...IS <characters>

The CLASS command comes in three different forms. The first (5) is used to define a synonym set, and it includes a description of members of
the set, and which string in the set is to be the canonical member used in translation down. The description of the set may be given by a regular expression, or by adding nonstandard form grammar productions (see Figure 11.)

CLASS...IS IGNORED

The second form of the CLASS command (6) is used to handle functionality mismatch during translation up. One option is to simply ignore the given nonstandard string.

CLASS...IS BRACKETED

The third form of the CLASS command (7) causes members of the class to be enclosed in brackets (or some other brace character, to use the parlance of the conceptual model), so that the string can be identified and dealt with during the use phase or other subsequent processing.

V.4.2 MDL and Translation Down

The second section of an MDL specification is used to describe the standard form grammar and how the constructs in the standard form are to be mapped to their nonstandard form equivalents. It is in this section that the homomorphism and permutation components of translation down and translation up are defined. In Figure 10, the symbols in uppercase represent start tags, and the symbols beginning with a "z" represent end tags.

Unaugmented productions

The expressions that follow each production define the value of the
Figure 10: An MDL Translation Down Specification
slspecdoc ::= zSPECDOC;

snafu ::= SNAFU text_stuff zSNAFU &ABORT;

text_plus ::= text_stuff text_plus |
| text_stuff
|
items ::= item items |
| item
|
item ::= LI text_plus zLI |
| (text_plus, "\n")
|


text_stuff ::= text |

section_plus ::= section section_plus |
| section
|

tex_text_or_subs ::= tex_text |
| sub_plus |

sub_plus ::= sub sub_plus |
| sub |

section ::= heading1 tex_text_or_subs |

heading1 ::= H1 text zH1 |
| (@newpage", "@section", "", text, ")", "\n") |

sub ::= heading2 tex_text_or_paras |

tex_text_or_paras ::= tex_text |
| paragraph_plus |

paragraph_plus ::= paragraph paragraph_plus |
| paragraph |

Figure 10, continued
Figure 10, continued

nonstandard form string to be generated when that standard form
construct is recognized during translation down. If no expression is given
by the TE (9), a default is generated by the TW. This default expression
defines the nonstandard form string for this construct as the concatenation
of the nonstandard form strings for each of the right-hand side symbols.

Productions with expressions

In these productions (10), the expression that defines the generated
nonstandard form string is allowed to refer to symbols from the right hand
side of the standard form grammar production, arbitrary nonstandard form
character constants, and synonym set names. The character constants and
synonym set names are used to define the homomorphism component of
translation. When a synonym set name is invoked (14), the canonical
member of the synonym set is actually used in the translation. The right
hand side symbols, character constants, and synonym set names may be
used in any desired order, thereby defining the permutation component of
translation down. (No such re-ordering was needed in this example.)

Productions with C code

It is occasionally useful to add code to the specification of TD (11) in
order to issue informatory messages or calculate values to be used later in
the translation. This code is copied into the code generated by the TW,
but is otherwise ignored.

&BEGIN context and &END context

The TE uses these commands (12,13) to indicate the start and end of
contexts, during which certain strings (designated by the
NONSTANDARD...IS SPECIAL IN command) have special meaning.

&IGNORE, &PASS, &WARNING, and &ABORT

The IGNORE, PASS, WARNING, and ABORT clauses are used to
handle functionality mismatch during translation down. We have added a
nonsense tag, <SNAFU>, to the standard form in order to illustrate the use
of these constructs (15). If a SF production has no meaning in the NSF,
the action to be taken during translation down depends on what the TE
specified. The IGNORE clause causes the string to be ignored. The
PASS clause causes the SF string to be passed through to the NSF object
unchanged. The WARNING clause is the same as the PASS clause,
except that a warning message is also generated. The ABORT clause
aborts the translation. Incidentally, the ampersand is used at the
beginning of these keywords to make it easier for the TW to distinguish
them from other MDL constructs.
V.4.3 MDL and the Nonstandard Form

In the third section of an MDL specification, the Translation Engineer provides information pertaining to the nonstandard form grammar. As we mentioned earlier, we cannot assume that the TE has access to a complete grammar for the nonstandard form. However, given their expertise with BNF and the nonstandard form, it not unreasonable to ask him or her to produce small NSF sub-grammars to help describe the nonstandard form. An example of this arises in handling synonym resolution during translation down. Recall that synonym sets are defined in the Declaration section using the CLASS...IS command, and that a regular expression defining the set may be included in the CLASS...IS command. However, since the synonym set may not happen to be regular, MDL provides a mechanism for defining a synonym set with nonstandard form grammar productions. These productions are restricted, in that the only symbols that can appear on the right hand side are nonstandard form character constants and the predefined symbol "text", which refers to a nonstandard form character string.

This same mechanism may be used to handle functionality mismatch during translation up, since the &IGNORE, &PASS, &ABORT, and &WARNING constructs, used in functionality mismatch during translation down, may also be used on nonstandard form productions.

\begin{verbatim}
NONSTANDARD FORM
15 centered_stuff ::= "@center(" text ")" (text);

style_stuff ::= "@style(" text ")"
               (text);
\end{verbatim}

Figure 11: Using MDL to Describe the Nonstandard Form
We have now described the syntax of MDL, and explained its semantics in terms of the user model. At the moment, in order to demonstrate the feasibility of our approach, we plan to use MDL in an ordinary edit, translate, and execute manner. Therefore, although this may not be true in the future, we don't expect MDL to have any special needs at the lexical or presentational levels.

V.5 Discussion

MDL is unique in that it was derived from a conceptual model of data translation, a model which is both quite general and free of implementation details. Furthermore, as a result of how it was derived from characterizations of the user as well as this conceptual model, each MDL construct has meaning in terms of both the user and the conceptual model.

MDL also shares some of the drawbacks of other programming languages, special purpose or otherwise. First, we still expect the programmer (the Translation Engineer) to be human and therefore fallible. It is unlikely that anyone will ever be able to specify a non-trivial data translation in a single attempt. Software systems tend to evolve over time, and it is therefore possible that we may at some point see fit to modify the Chameleon architecture. However, since MDL was derived from the conceptual model and not from the Chameleon build phase, it is unlikely that changes to the Chameleon architecture will have a serious impact on MDL.

One might ask if there was a way to measure the distance between the conceptual model and the user, such that when this distance exceeded a certain quantity we would know that there was some aspect of the conceptual model that we just couldn't express in the user's vocabulary, which would imply that some of the functionality of the
conceptual model was inherently inaccessible. In the example above, it was clear that everything in the conceptual model could be expressed in the user's vocabulary. We did discover that our list of items in the user's vocabulary lacked the notion of nonstandard form subgrammar that we needed to handle synonym resolution and functionality mismatch during translation up. Since we assume the TE knows BNF and the nonstandard form intimately, this should not be a problem.

In this chapter, we have described a language and its semantics for specifying data translations. This language has been derived from the conceptual model given in Chapter IV, without regard to how it will be implemented, or any detailed knowledge of the build phase. In the next chapter, we describe the mechanisms that the build phase uses to instantiate the conceptual model.
Chapter VI
The Chameleon Architecture

In this chapter, we show how the Chameleon build phase is derived from the conceptual model described in Chapter IV. In particular, we describe how the entities in the conceptual model are manifested in the Chameleon system, and how the build phase handles the six situations we have identified. We describe the build phase in some detail, not so much because it is critical to demonstrating the feasibility of our approach, but to impress upon the reader the complexity from which we are shielding the Translation Engineer.

VI.1 Approach

In deriving the system from the conceptual model, we follow basically the same procedure we used in the previous chapter to derive the user interface. We first describe how the system instantiates each of the entities in the conceptual model, and then we describe how the system handles the six situations in translation up and translation down. In deriving the system to instantiate the conceptual model, we are not as constrained by vocabulary as we were during the derivation of the user interface: we assume that the designers of the system are well-versed in programming and computer science concepts. The only hard requirement is that the system implement the conceptual model faithfully. The question that we must now answer is, how do we specify and implement the translation mappings in the conceptual model?
During the development of the Chameleon project's early prototypes, before we had completed the formal model and discovered the principle of separate design of system and user interface, we observed that attribute grammars had some properties that would be useful in solving the data translation problem. Attribute grammars, or AGs, have long been used for purposes of translation, especially in the areas of compiler construction and programming language semantics [Knuth 68]. Since AGs are capable of computing any computable function, they are certainly capable of implementing translation up and translation down. Attribute grammar evaluators exist that allow certain large classes of AGs to be evaluated quickly. Finally, an algorithm that allows AGs to be inverted automatically has been invented by Yellin [Yellin 87]. The remainder of this chapter is devoted to showing that the conceptual model of data translation can be instantiated using AGs, and that an AG-based system can be designed that handles the functionality of the conceptual model.

VI.2 AGs and the Conceptual Model

We now present the entities in the conceptual model, along with descriptions of how they can be realized in an AG-based system.

standard form A context-free grammar describing a Braced language.

nonstandard form

A language that describes the structure of data objects used by a software package.

translation up A function computed by an attribute grammar that translates nonstandard form strings to the standard form.
translation down
A function computed by an attribute grammar that translates standard form strings to the nonstandard form.

start tag
A member of the set of brace symbols in the standard form that marks the beginning of a standard form construct.

end tag
A member of the set of brace symbols in the standard form that marks the end of a standard form construct.

nonbrace string
Text in a standard form document with no embedded start or end tags.

substitution
A function computed by an attribute grammar that translates a string into a set of strings, where the set may have zero or more elements.

homomorphism
A function computed by an attribute grammar that translates a string into a set of strings, where the set has exactly one element.

permutation
A function computed by an attribute grammar that associates a set of strings with another string by concatenating the member strings in any order, but using each string exactly once. In particular, translation down may be thought of as the function computed by an attribute grammar that translates each standard form object into an equivalent nonstandard form object, where this function being computed is the composition of a homomorphism and a permutation.
inverse substitution

A function computed by an attribute grammar that translates a given string into a set of strings such that each of the members of the set of strings is translated to the given string by a certain substitution. In particular, translation up may be thought of as an attribute grammar that translates each nonstandard form string into a set of standard form strings such that those standard form strings are translated to the given nonstandard form string by a translation down attribute grammar.

VI.3 The Six Situations - A System Perspective

In order to discuss how the six situations are handled in an AG-based system, we shall assume that we have two AGs, one implementing translation up, and the other implementing translation down. The translation down AG consists of a context free grammar for the standard form, augmented with semantic functions that show how each of the productions on the standard form grammar correspond to strings in the nonstandard form. These semantic functions actually define the homomorphism and permutation components of translation down. The translation is actually performed during a parse of the standard form object. As the object's parse tree is built, the value of a designated semantic function, which we call the trans attribute, is calculated at each interior node in the parse tree. When the parse is complete, the value of the trans attribute of the root node is the result of the translation, namely, a complete nonstandard form string equivalent to the original standard form object.

The translation up AG consists of a CFG for the nonstandard form, and the productions in this grammar are also decorated with semantic functions that compute the value of the trans attribute. When this AG is used to parse a nonstandard form
object, the value of the translation attribute at the root node of the parse tree is the value of the corresponding standard form object.

So, we would hope to be able to reduce the problem of data translation to the problem of constructing appropriate translation up and translation down attribute grammars. Unfortunately, there is no guarantee that the Translation Engineer has a context-free grammar for the nonstandard form available, and without such a grammar, we cannot build a translation up AG.

The solution to this problem lies in the recent invention of an algorithm for inverting attribute grammars. Then, if we build an AG implementing translation down, we can get an attribute grammar implementing translation up very cheaply. It is appropriate at this point to give a general overview of the inversion algorithm. (For a more detailed description, see [Yellin 86] or [Yellin 87].) Recall that an attribute grammar is a context-free grammar, in which the production rules are associated with attributes. Suppose we have an attribute grammar that performs some translation function on its input, and that it uses a certain attribute, the trans attribute, to compute the value of the translation. If these trans attributes satisfy certain restrictions, then they can be regarded as CFG productions in their own right. Basically, the trans attribute must be defined in terms of what are called token permuting functions, which means that they simply re-arrange the symbols on the right hand side of the production, without adding or deleting information. If the AG satisfies this condition, then the entire set of trans attributes can be regarded as a CFG, and the resulting grammar will define the language consisting of strings in the range of the original translation function.
We now describe how each of the six situations can be handled in a system in which an attribute grammar implementing translation down is inverted to produce an attribute grammar implementing translation up.

**TD: Functionality Mismatch**

A standard form production for which the trans attribute has no reasonable value.

**TD: Homomorphism**

A standard form grammar in which each standard form production has a semantic function that produces a single string representing a translation to the nonstandard form.

**TD: Synonym Resolution**

In this case, a number of nonstandard form strings exist that would be reasonable translations of the standard form construct. We rely on the Translation Engineer to tell us which of these strings should be used as the value of the translation. However, we also supply a description of the nonstandard form strings that could also have been used. Then, during translation up, nonstandard form strings that fit this description can also be mapped back up to the original standard form string. This description of semantically equivalent nonstandard form strings will take the form of a fragment of the nonstandard form grammar, and we will add this grammar fragment to the NSF grammar generated by the inversion algorithm.
TU: Functionality Mismatch

The inversion algorithm assures us that a trans attribute will be assigned to each production in the NSF grammar it generates. However, it is possible that the function computed by the translation down attribute grammar is into, in the sense that not every nonstandard form object is in the range of the translation down function. If such a string is presented to the translation up AG, it will not be able to recognize it as a valid nonstandard form object, and the result will be analogous to a syntax error in an ordinary high-level language program. It will then be up to the TE to refine the translation up information in the MDL file, in order to do a complete job of recognizing nonstandard form strings.

TU: Inverse Substitution

In this case, the attribute grammar generated by the inversion algorithm associates a single standard form string with each production in the nonstandard form grammar. The value of the root node’s trans attribute is the result of translation up.

TU: Ambiguity

When the translation down attribute grammar is inverted, the result is an attribute grammar that maps nonstandard form objects back up to the standard form. Recall that we use the term ambiguity to refer to situations in which a single nonstandard form object could be mapped, by translation up, to two (or more) distinct standard form objects. This may happen if the translation up attribute grammar (or, more precisely,
its underlying CFG) is ambiguous. We will be able to detect ambiguity in the nonstandard form grammar when its parsing tables are computed. It is also possible that individual character sequences (i.e. tokens) in the nonstandard form object have more than one meaning, depending on context. We rely on the TE to indicate when a character sequence may be interpreted in more than one way, and which token value is to be used under which circumstances.

In this section, we have seen that a system based on invertible attribute grammars can be designed to faithfully implement the conceptual model of data translation. The rest of this chapter is devoted to a description of just such a system, namely the Chameleon build phase.

VI.4 The Build Phase of the Chameleon Architecture

The Chameleon build phase is one of the two major components of the Chameleon architecture. The purpose of the build phase is to aid the Translation Engineer in the construction of the translation up and translation down software. These routines are then employed during the use phase, the other major component, to actually translate data objects. Figure 12 shows the overall structure of the Chameleon build phase. Chameleon was designed particularly for the translation of electronic documents, although its basic components are useful in any domain. Several other components of the build and use phases, especially the SGML compiler shown in Figure 12, are specific to the electronic document domain, and wouldn't necessarily be used in other domains.
Figure 12: The Chameleon Build Phase
Build phase processing starts with an SGML document type definition. SGML, which stands for Standard Generalized Markup Language, is a language for writing electronic document standard forms [SGML 86]. A particular SGML document type definition, then, can be regarded as a grammar for a particular standard form. The SGML Compiler converts this grammar into a format compatible with the Translator Writer. Figure 13 is an SGML document type definition for a class of simple documents for writing software specifications.

```xml
<!ELEMENT specdoc (section+)>
<!ELEMENT section (heading1, (technical_text | subsection+))>
<!ELEMENT technical_text (#CDATA | list | code)> *
<!ELEMENT subsection (heading2, (technical_text | paragraph+))>
<!ELEMENT paragraph (heading3, technical_text)>
<!ELEMENT code (#CDATA)>
<!ELEMENT list item*> *
<!ELEMENT item (#CDATA)>
<!ELEMENT (heading1, heading2, heading3) (#CDATA)>
```

Figure 13: An SGML Document Type Definition

This SGML definition was run through the build phase SGML Compiler, and the result was a grammar for that standard form, in a Yacc format. This file was then augmented with MDL commands, and an excerpt of the resulting specification of a down translation, from the standard form to Scribe, was shown in Figure 9.
The Translation Engineer uses the TW to decorate the standard form grammar with MDL statements that describe how standard form objects are to be translated down to a target nonstandard form. The output of the TW consists of three files: an invertible attribute grammar, and scanners for the standard and nonstandard forms. The invertible AG is compiled directly into a Translation Down routine by the Down Translator Generator. This same invertible AG is also given to the Inverter, which converts it into an AG that maps the nonstandard form back up to the standard form. The output of the Inverter is given to the Up Translator Generator, and that in turn produces a Translation Up routine. The standard form scanner describes the standard form at the lexical level. The nonstandard form scanner, used during translation up to tokenize the nonstandard form document, includes a description of what tokens exist in the nonstandard form, and a description of those synonyms that can be resolved at the lexical level.

In this chapter, we have seen how the Chameleon build phase was derived from the conceptual model. We have now completed the process of separately deriving the system and user interface from the conceptual model. The next step is to make sure that the user interface that we have designed will actually fit together with the Chameleon build phase in order to form a cohesive unit that the Translation Engineer can use to access the functionality of the conceptual model.
Chapter VII
Using the Translator Writer

In the previous two chapters, we showed how the design of a user interface and the design of a system could be derived from a conceptual model. In this chapter, we show how these two designs can be connected, resulting in an integrated system that implements the functionality of the conceptual model, and gives the user complete access to that functionality. We then describe how the Translator Writer was used in several significant data translation applications.

VII.1 Connecting the System and the User Interface

At this point, we need to assure ourselves that the user interface we have derived from the conceptual model is indeed compatible with the system that was derived from that same model. Compatibility in this sense means that the user interface provides the system with whatever information must come from the user, including specifically what the system is to do and any data required to do it. This requires us to identify the various types of data the build phase requires from the user in order to implement the functionality of the conceptual model, and then to make sure that the user interface, namely the Translator Writer, provides that information.

We know from the description given in Chapter V that MDL is capable of expressing all the information that the conceptual model, and therefore any system
properly derived from it, may demand from the Translation Engineer. In particular, we know that MDL is capable of expressing all the information needed to specify translation up and translation down. The TW accepts this information from the Translation Engineer, and then examines the MDL specification to collect information that will be needed by the other components of the build phase. This information is then presented to the build phase in the form of three input files:

- A Yacc file that implements the Translation Down AG. This file describes the syntax of the standard form and nonstandard form, and how the Translation Up and Down mappings are done. To produce this file, the TW uses information from the expressions following each standard form production (including the TD functionality mismatch commands such as &IGNORE). This AG is compiled to produce Translation Down. The same AG is passed to the Inverter, which builds an AG describing Translation Up. This AG may contain information about how to handle TU functionality mismatch, or nonstandard form grammar productions that describe synonym classes. This AG is then compiled to produce the Translation Up routine itself.

- A Lex file that is used to generate the standard form scanner. This file describes the tags defined by the standard form. When building a group of translators, mapping between a single standard form and a number of nonstandard forms, the standard form scanner files will be very similar. A particular standard form scanner will be different from the others only because of the fact that if the standard form and that nonstandard form use different character sets, the standard form scanner will describe how the standard form characters are to be mapped to their equivalents in that particular nonstandard form. Information from the STANDARD FORM
command is used in the construction of this file, which is compiled along with the Translation Down AG to produce the Translation Down routine.

- A Lex file that is used to generate the nonstandard form scanner. This file includes information pertaining to how synonym resolution and ambiguity are to be handled during Translation Up. It also handles any character mapping needed during Translation Up if the standard form and nonstandard form use different character sets. The TW uses information from the KEYWORD, CHARACTER, and CLASS commands to produce this file, which is compiled along with the inverted grammar to form Translation Up. The CHARACTER and CLASS commands are used to produce character mapping and synonym resolution information, respectively. The KEYWORD command is used to detect a certain class of ambiguities that may arise during translation up. Text formatters tend to have special characters that mark the beginning of commands and keywords. The KEYWORD command identifies these special characters to the TW, which makes sure that the nonstandard form scanner doesn’t confuse formatter commands with ordinary text.

The amount of effort that needs to be expended in making sure that the interface really fits the system probably depends (at least in part) on the complexity of the input data structures required by the system. In our case, the build phase requires three input files, each having a rather complicated structure. It is easy to imagine situations in other domains in which just as much (or perhaps more) effort would need to be expended to produce a large number of relatively simple input data structures, consisting perhaps of small input files or messages.
The Translator Writer, including the MDL language processor and the routines that transform information from the MDL specification into a form in which it can be used by the other components of the build phase, has been implemented in a UNIX™ environment. The MDL language processor was written using Yacc and Lex, and the other routines were written in C.

VII.2 Using the Translator Writer: A Scenario

It is appropriate at this point to describe how one goes about using the TW and the Chameleon build phase to build a pair of translators which implement Translation Up and Translation Down to and from a certain nonstandard form. The first task is to create or acquire a BNF description of the standard form grammar. Remember that it is not specifically the Translation Engineer's responsibility to create the standard form grammar, but we do assume that he or she is able to read the standard form grammar and understand what it means.

After the standard form grammar is ready, the Translator Writer is used to specify the translation. There are three distinct steps in this process:

1. The standard form grammar is augmented with details of how a given standard form string, supposedly representing a single article, is to be translated down to a particular nonstandard form. This is done by adding MDL commands to the standard form grammar with a text editor. In this way, the Translation Engineer indicates how each of the six translation sub-tasks is to be handled: The TE specifies how each standard form construct is to be mapped to a corresponding nonstandard form construct, and how the various instances of functionality mismatch, and synonym resolution, and ambiguity are to be handled. In many
cases, the standard form production may be mapped down to the nonstandard form by simply specifying appropriate character constants. In many other cases, involving productions whose purpose is to show the order in which standard form constructs may occur, no specification is needed at all. This is also the time when the TE may add productions that describe portions of the nonstandard form grammar. There are three reasons to do this: to define a synonym class, to define a set of nonstandard form strings that are to be treated alike with regard to functionality mismatch, or to stipulate that certain nonstandard form constructs are to be mapped to specific standard form constructs, as a way of partially resolving (or preventing) ambiguities. (Except for the KEYWORD command, there seems to be little that can be done at this point to resolve ambiguity statically, a point that will be discussed further when we describe our new insights into the architecture itself.)

2. The augmented grammar (then known informally as the MDL file) is processed through the Translator Writer. If the MDL file is syntactically valid, the TW will do some checking to make sure the standard form grammar and the translation specification meet the conditions imposed by the build phase components. The TW makes sure, for example, that all the nonterminal symbols appear on the left side of a production at least once, a condition imposed by Yacc. The TW also makes sure that the conditions for invertibility are satisfied. If the MDL file passes these checks, the TW writes the files that drive the various build phase components. If the MDL file fails these checks, the TW issues appropriate messages. These messages are all phrased in terms the TE will understand. Suppose, for example, that the TE formulates an MDL
expression that would cause the system to discard the information associated with one or more of the symbols on the right side of a standard form grammar production. In this example from the troff grammar in Appendix B, the appendixmatter symbol appears on the right side of the standard form production, but has been omitted from the corresponding MDL expression.

```plaintext
article : ARTICLE frontmatter bodymatter appendixmatter backmatter ARTICLEz
    (font_stuff1,"\n", font_stuff2,"\n",
eqn_init1,"\n", eqn_init2,"\n",
eqn_init3,"\n", eqn_init4,"\n",
frontmatter, bodymatter, backmatter)
```

The resulting AG for Translation Down cannot be inverted, since there would be no way to construct an AG for Translation Up that could recreate the lost information. The most important condition of invertibility is that each right side symbol is mentioned exactly once in the corresponding MDL expression, so in this case the TW issues a message informing the Translation Engineer of the need to mention the missing symbol(s) in the MDL expression. In the example given above, this message was

**61 Error: appendixmatter must appear exactly once in this expression.**

where the 61 refers to the line number in the MDL file. The Translation Engineer could then use an editor to type "appendixmatter" in the proper place in the expression. If appendixmatter indeed had no meaning in the nonstandard form, the &IGNORE clause would be used in the production defining appendixmatter, in which it appears on the left side.

(Other messages are issued by the TW if certain internal table sizes are
exceeded. However, these messages are not directed to the Translation Engineer, but to the person responsible for the maintenance of the TW itself.)

3. As soon as the TW has produced the necessary files, build phase processing is started. The final output of the build phase consists of two translators: one for Translation Up, and the other for Translation Down. These translators may then be tested on sample standard form and nonstandard form documents. If the output from the translators is unsatisfactory, which would happen if some information supplied by the TE is wrong or inadvertently omitted, the TE can go back to the MDL file, modify it, and re-build the translators. This may happen, for example, if the TE remembers some other nonstandard form constructs that should belong to a certain synonym class, but were omitted from the description of that class.

VII.3 Using the TW to Build Production Quality Translators

We have now completed the description of our approach, and described how the TW is used by a Translation Engineer. In this thesis, we have argued that our idea of separately deriving a system and its user interface from a conceptual model is reasonable, and we have demonstrated that it is workable by actually using it to build the user interface for a non-trivial system. This interface, the Translator Writer, has been used by several people to perform data translation tasks. We now need to substantiate our claim that separately deriving the system and the user interface from a conceptual model does indeed result in more accessible software than the traditional approach. In this section, we relate our experiences in building translators, both with
and without the use of the Translator Writer. We then indicate what these experiences tell us about the merits of our approach in comparison with the traditional approach.

VII.3.1 Experience in Building Translators

We first describe our experience in specifying translations without the use of the Translator Writer, and then we describe our experiences in which we did use it. In both efforts, the standard forms used were versions of one supplied by the American Association of Publishers for the markup of articles.¹

Two years ago, we built a prototype translator for Chemical Abstract Services that translated the AAP standard form down to a proprietary nonstandard form called J124. Building this translator, using Lex, Yacc, and C, took two people, working at 75% time, about two months to build, making a total of three person-months. Since this translator was uni-directional, implementing Translation Down only, we didn’t worry about ambiguity or functionality mismatch in the other direction, or the need to recognize the members of each synonym class so they could be mapped back up to the same standard form construct. Furthermore, since we didn’t need to use the Inverter to produce a Translation Up routine, we didn’t need to make sure that the Yacc file was invertible. To implement Translation Up would have certainly added another person-month or two to this effort.

We decided to test the Translator Writer, as well as several other build phase components, by building translators that would map several well-known text formatters to and from a standard form. The formatters that we were most interested

in translating to and from this standard form were Scribe, LaTeX, and troff. The TW has been used to describe translations down to Scribe, troff, LaTeX and WYSIWYG (what you see is what you get) formats.\(^2\) A complete MDL specification for SGML to troff is given in Appendix B.\(^3\)

The translators for Scribe, LaTeX, and troff that we built using the TW and the Chameleon build phase are similar in complexity to the CAS effort, in the sense that the AAP standard forms used are quite similar, and the nonstandard form grammars involved are of comparable size. To build these translators took three people, working at 33% time, about three months, or roughly one person-month/translator. Two of these three translators (for Scribe and LaTeX) were written by people who were not involved in either the CAS effort or the design and construction of the build phase.\(^4\) These figures indicate that the TW appears to increase the productivity of Translation Engineers by a substantial amount.

During the construction of our translators, we found that the most serious problems were related to the sheer size and complexity of the AAP article standard form. Another factor was our relatively limited experience with the troff, Scribe, LaTeX, and WYSIWYG packages; we spent a considerable amount of time studying

\(^2\)Thanks are due to the Chameleon project members who acted as the Translation Engineers for these efforts: Julie Barnes for Scribe, Joan Bushek for LaTeX, and Craig Joseph for the WYSIWYG format. The author acted as the Translation Engineer for the troff effort.

\(^3\)The troff, troff, eqn, and tbl packages are described in *Formatting Documents on the Sun Workstation*, and *Using nroff and troff on the Sun Workstation*, both published in 1986 by Sun Microsystems, Inc., Mountain View, California.

\(^4\)These figures are not convincing from the point of view of statistical rigor; to amplify on a point mentioned in Section III.1.3, statistically valid results in this area are very hard to come by because of numerous and challenging methodological problems.
the features of these packages in detail. In comparison with these two factors, the effort required to actually learn and use the Translator Writer to specify the mappings from standard form strings to nonstandard form strings was quite small. Given smaller standard form grammars, it is quite possible to build working translators in a matter of hours. The troubles encountered by Translation Engineers during the specification and production of these translators were due to the complexity of the standard form, or to idiosyncrasies in the nonstandard forms. It would seem that, as a result of using the Translator Writer, the TE can spend less time learning how to specify translations, and devote more time to actually creating and refining them.

When we were building the CAS translator, we had nothing more than the UNIX shell to serve as the user interface to our system. The input files used by the system were built by hand. When we built our new translators, we had a more elaborate user interface, namely the Translator Writer, which seemed to provide improved access to the desired functionality. How do we know that it is the approach that we used to build the Translator Writer that is responsible for this improved access, and not just the fact that we had a better user interface? To put this question another way, how do we know that our approach to user interface design results in more accessible software than the traditional approach?

As we stated at the outset of this thesis, the point of this research was to show how separately deriving a software system and its user interface from a common conceptual model can result in more accessible software than the traditional approach. We have shown that our approach is reasonable, and we have described how it was

---

Although our methodology assumes that the TE is already intimately familiar with the nonstandard form.
applied in the development of a real system. The evidence we have presented so far indicates that the TW gives better access to the desired functionality than the (minimal) interface that we had during the CAS effort. This evidence on its own does not prove that our approach will result in more accessible software than if the traditional approach had been employed under the same circumstances. However, as a result of the experience we gained by using the approach, we are prepared to argue that our approach will result in a more accessible user interface for the Chameleon build phase than the traditional approach. In the next subsection, we present this argument.

VII.3.2 Arguing the Competitiveness of our Approach

Let us suppose that a group of people as qualified as ourselves built a user interface for the Chameleon build phase, called Q, using the traditional approach. We can then arrange the three interfaces we've mentioned on a scale, depending on how well they meet our accessibility criteria, i.e. how well they avoid the accessibility problems of inappropriate vocabulary, inappropriate syntax, and distortion of the desired functionality. We will assume, for the sake of argument, that some fair way of assessing how well each interface satisfies these criteria is available. If two interfaces do equally well at avoiding these problems, we will judge the one that was cheaper to develop to be superior. Clearly, the user interface we had for the CAS experience is much nearer the bottom of this scale than Q or the Translator Writer. The Translator Writer does avoid these problems, so it would rank closer to the top. Where is Q likely to fall on this scale?

We grant that it would be possible for Q to have avoided contamination from system vocabulary and syntax just as well as we do, so Q might be competitive with
the TW with respect to these two factors. However, even if the developers of Q had used SUPERMAN or some similar technique, they would have risked going through some number of iterations before coming up with a competitive interface, since in the traditional approach there is the potential for system contamination. In our approach, we avoid these problems from the start. So, even if Q did just as well as the TW in terms of avoiding system contamination, we believe that it would be difficult for Q to be competitive with us in terms of development cost.

We can now assume that user interface Q has avoided the two problems of inappropriate vocabulary and syntax; otherwise, as we just showed, the TW comes out ahead. We now consider the question of how Q compares with the TW in terms of the third criterion, avoiding distortion of the desired functionality. If the developers of Q were using the traditional approach, then we can show, using examples from our own experience, that it is in fact highly unlikely that Q provides undistorted access to the desired functionality, or if it does so, then only at great expense. If the developers of Q claim that their interface avoids distortion of the desired functionality, then we have two questions for them:

1. How do you know that Q provides access to all the desired functionality? In our approach, the conceptual model says exactly what functionality should be included in the user interface. Using the traditional approach, (and, in particular, lacking an abstract conceptual model to guide them) the developers of Q would have only had two things to work with while designing the user interface: a description of the structure of the build phase's input files, and a description of the system's functionality in terms of what it did with those files. (As we mentioned earlier in this chapter, the TW produces three such input files:
a standard form scanner in Lex format, a nonstandard form scanner in
Lex format, and an invertible AG in Yacc format.) Using the traditional
approach, and given this information, the developers of Q would have no
way to know that Q was in fact asking for all the information it should
be asking for. How, for example, would they know to ask for
information about synonym resolution? Without a conceptual model to
guide their effort, it might take many iterations before they discovered
the need for this functionality. Granting that, through experience, they
would eventually discover this and the other five cases, how would they
know that there weren't still more problems waiting to be discovered?

2. How do you know that Q doesn't provide any excess functionality? The
conceptual model not only says what functionality should be included in
a user interface, it also says what should not be included. The build
phase itself, and the files it takes as input, are complicated enough
already. Using the traditional approach (and, again, lacking a conceptual
model), the developers of Q might be tempted to include too much
functionality, functionality that isn't or shouldn't be provided by the
system. This, in turn, would result in a user interface that is needlessly
complex, thereby hindering the user from doing what he or she wishes to
do.

Here is a specific example of this extra functionality: As a result of
using our approach, we became aware of the distinction between
problems that pertain to translation per se (i.e. mapping of strings), and
those that pertain to recognizing the various standard form and
nonstandard form constructs that are to be translated (i.e. recognizing
strings). Lexical analysis of the nonstandard forms has proven to be especially challenging, due primarily to the idiosyncrasies or excessive flexibility of the various nonstandard forms. It turns out to be much easier to handle these problems in pre- and post-processors, and to avoid doing such things as part of translation itself. One example of this excess flexibility is the wide variety of ways in which Scribe commands can be capitalized (e.g. @begin, @Begin, @BeGîN, etc.) It is of course theoretically possible to write an AG that recognizes all these variations, but such a grammar would be needlessly complicated. A user interface that allowed the user to specify such a recognition algorithm would also be needlessly complicated. In this case, it would be much easier to write a small pre-processor (a small program, or perhaps a command file that invokes the EMACS replace command) to enforce some uniform capitalization scheme. In our case, having a conceptual model made us more confident that splitting this processing off into separate components was, in fact, the right thing to do. However, lacking a conceptual model to guide them, it is unlikely that the developers of Q would have avoided problems like these without first expending considerable effort through iteration.

We have just argued that, by using the traditional approach, it would be difficult to construct a user interface to the build phase that is as accessible as our Translator Writer. Even if an interface that was just as accessible could be developed, it seems unlikely that this could be done without incurring substantially greater development costs.
VII.3.3 Other Benefits of this Approach

Not only did the approach work from the standpoint of accessibility, but it also resulted in two other benefits. First, from a purely practical point of view, the approach literally made it possible for us to build some non-trivial translators. This allowed us to gain insight into the Chameleon architecture that we would never have attained otherwise. We discuss three of these insights here, and more will be said on this subject in the next chapter.

1. Recall that ambiguity arises when a nonstandard form construct could be mapped to more than one standard form construct. When this is manifested in an attribute grammar, the result is a reduce/reduce conflict. (The name comes from the fact that Yacc's LALR(1) parser can enter a state in which it can reduce by two or more distinct productions.) When it discovers such a conflict, Yacc will reduce by the production that it encountered first. This may not be the Translation Engineer's intent, however, so the generated Translation Up routine will be unreliable. We now see that there are two ways to handle these conflicts. One way, which the build phase uses now, is to have Yacc postpone the decision until a human being can make it during the use phase. The other way is to give the TE the ability to remove from the nonstandard form grammar productions that cause reduce/reduce conflicts. We are now deciding how MDL and the build phase should be changed to do this, and one of the questions we're considering is how to describe this functionality using the TE's terminology.

2. Our second insight is that we now understand that there are circumstances in which it is better to use the Inverter to produce a first
version of Translation Up, which can then be developed on its own, independent of what may happen to the Translation Down AG from which it was first produced. We don't yet fully understand the ramifications of this, but we have already seen some of its benefits. First, when working with large standard forms and nonstandard forms, the time it takes to actually build a translator can be substantial. Currently, the only way to make a change in a translator is to rebuild Translation Down as well as Translation Up, even if the change only applies to one of them. If the TE can manipulate the Translation Up AG directly, and if we make the strong assumption that he or she knows enough to do this safely, this can save time during the build phase. By using this same technique, we were able to make the Translation Up component of the WYSIWYG translation completely automatic, i.e. both translators can run without human intervention, making it unique among our translators in this respect.

3. We also understand more about the use of context information in computing the translation mappings, for which Yacc is not especially well suited. Context information could be used in at least two ways by our architecture:

- To state that certain productions are only to be reduced under certain circumstances. This, in turn, would provide another way to deal with ambiguities.

- To compute the canonical member of a synonym set based on context, rather than fixing its value beforehand.

To see why this second feature would be helpful, we draw again on the
troff example. Suppose that we are trying to produce numbered or bulleted paragraphs inside lists (corresponding to Scribe's \texttt{@enumerate} and \texttt{@itemize}, respectively.) The standard form stipulates that paragraphs inside lists are no different from ordinary paragraphs, and the canonical member of the synonym set for paragraph specifies block style with no indentation or heading. Unfortunately, this means that even when a bulleted or numbered paragraph is desired, there is no way to change the troff command produced by Translation Down. Fortunately, in the translators we have built to date, lacking this functionality has proven to be only a minor inconvenience. We are now in a position to decide whether we should enhance Yacc, or acquire some other attribute grammar evaluator, such as Linguist [Farrow 82], which might be more appropriate for such purposes.

A second benefit of this approach is that the conceptual model, and therefore the user interface derived from it, enforces a common framework for thinking about and discussing data translation. This way of thinking has led us to deeper insights about data translation itself, and is especially helpful when problems arise during the specification of translators. These insights have enabled us to write some guidelines for use in the definition and design of standard forms and nonstandard forms (e.g. text formatters) [Mamrak 88]. Some of these guidelines are:

- We advise those who define standard forms to resist the temptation to include tags that carry formatting information. Standard forms express the meaning of objects; there is no place in a standard form document for local processing details. We also advise against the use of certain notational features that apparently shorten the standard form specification,
but which are in fact prone to result in cumbersome or incorrect standard form grammars.\footnote{SGML has several such features, including inclusion exceptions, exclusion exceptions, and the \& operator.}

- We advise those who design nonstandard forms and the text formatters that process them to avoid extraneous flexibility of the sort seen in Scribe's command capitalization scheme. We also encourage them to use special characters that set off commands or keywords in a consistent manner. Scribe, for example, uses the "@" symbol in front of all of its keywords and other commands - except within the @math environment. This means that Translation Up has to be prepared to recognize words such as FROM and TO as ordinary text almost everywhere in the document; everywhere, that is, except within a math environment, where they are keywords.

In this chapter, we have described how the last step of our methodology, reconciliation of the system and the user interface, was done in the development of the Translator Writer. We then related our experiences in building translators, both with and without using the Translator Writer. This experience has given us evidence and arguments to support our thesis statement, namely, that separately deriving a system and its user interface from a common conceptual model results in more accessible software than the traditional approach. In the next chapter, we summarize our contributions and give directions for future work.
Chapter VIII
Conclusions and Future Work

VIII.1 Conclusions

A system is said to be *inaccessible* if the user for some reason finds it hard to access, or use, the desired functionality. Many large systems have been designed, built, and installed, only to be found to be inaccessible by their intended users.

We have shown how the use of inappropriate vocabulary or input language syntax can lead to accessibility problems, and how the use of such inappropriate vocabulary or syntax can arise from the traditional software development process. This thesis has described an alternative to the traditional software development process designed to avoid the accessibility problems found in complex computer software systems. Specifically,

- We have proposed that a system and its user interface should be designed in isolation from each other, and that both of these should be derived from a conceptual model of the desired functionality.

- The conceptual model of data translation summarized in Chapter IV is the first formal characterization of the data translation problem that accounts for every situation that can arise during the translation process. This conceptual model was then used to drive the design of the Translator Writer and the rest of the Chameleon build phase.
• We have demonstrated the feasibility of this approach by using it in the construction of the Chameleon architecture, a system that supports the specification, construction, and use of data translation tools.

In this thesis, we have described how this new approach was used in the design of the user interface to the Chameleon build phase, known as the Translator Writer. We have presented evidence, based on our experiences in building translators, to support our statement that the Translator Writer is more accessible than a similar interface, developed using the traditional approach, would have been.

VIII.2 Other Questions

Several questions remain to be answered. The first few questions deal with the application of the approach to data translation, and the rest deal with the approach per se.

VIII.2.1 The Translator Writer as a Software Tool

Several specific improvements could be made to the TW and the build phase. One of the problems with the TW is that it is not capable of detecting TU ambiguities on its own, especially when these ambiguities are manifested as conflicts in the generated attribute grammar for translation up. Even if a grammar for translation down has no conflicts at all, the inverse grammar may have many thousands of them. (At one point in the development of the troff example in Appendix B, the grammar for TD had 2 shift/reduce conflicts, and the inverse grammar had about 17000 shift/reduce and more than 3500 reduce/reduce conflicts.) Reduce/reduce conflicts are especially serious, since they are directly related to the ability of translation up to recognize and
re-arrange components of the nonstandard form object. Correcting these conflicts is also a considerable inconvenience for the Translation Engineer, for two reasons. First, determining the exact cause of a reduce/reduce conflict is not always easy, even for people with a background in LR parsing theory. Second, once a conflict has been found and corrected, the TW, Inverter, and Up Translator Generator must be re-run. For a large grammar, such as the one in Appendix B, this can take ten or fifteen minutes on an (otherwise) lightly loaded Sun 3 workstation. If an interface to the Up Translator Generator was developed that allowed such conflicts to be reported to the Translation Engineer directly, this would make it easier to develop MDL specifications. If the interface could also supply suggestions on how to modify the MDL specification to eliminate the reported conflicts, that would be a significant contribution. I know of no compiler construction tool that gives the developer any automatic assistance in resolving conflicts in grammars. The integration of the TW, Inverter, and Translator Generators into a single (albeit rather large) program would also make it worthwhile to develop a user interface that was easier to use from an ergonomic (i.e. syntactic or lexical level) perspective.

The model of permuting functions in Chapter IV is more general than the permuting functions used by the TW. Specifically, the permuting functions that can be specified using MDL are a proper subset of the permuting functions, because only those symbols that appear on the right side of a given standard form grammar production are allowed to be arguments of a permutation function defined for that production. In order to make the TW handle permuting functions in their most general form, this restriction would need to be relaxed. However, making this permutation function facility too general might result in the inadvertent specification of functions that are not automatically invertible, or perhaps not even one-to-one.
VIII.2.2 Possible Extensions or Improvements to Our Approach

We have demonstrated the feasibility of this approach by using it on a particular conceptual model that describes a particular kind of programming activity. Is it possible to characterize those conceptual models that are especially appropriate, or especially inappropriate, for use with this approach? For example, is it reasonable to use conceptual models that are in some sense incomplete, meaning that no analysis yet exists that describes all the entities and events that reside or occur in that model? To answer questions like these, it would be necessary to apply this approach in a variety of domains with different sorts of conceptual models.

Just as one can ask about different or improved conceptual models, one can ask about different or improved ways of characterizing the system’s intended users. We have chosen a relatively simple method, based on what we assume about the user’s vocabulary, as a means of characterizing the Translation Engineer. How much (if any) benefit could be derived from more sophisticated characterizations of the user? For example, we might be told that the user is more familiar with some concepts than others, and that the user could be expected to do a certain amount of learning. Then we might be able to develop some way of measuring the effectiveness of our approach given a conceptual model and a certain user profile.

Can this approach be used to assure any other desirable properties of software? One aspect of software quality that is closely akin to accessibility is communicativeness. Just as a system’s accessibility refers to the ease with which the user gives information and direction to the system, communicativeness refers to the amount and usefulness of the information that the system gives back to the user. This approach shows promise with regard to this communicativeness property in two
respects. First, it should be possible to identify, in the conceptual model, the types of information the user may need to see. The system could then be designed to provide this information, and to phrase messages to the user in terms that the user can understand.

One final question has to do with how this approach, which we have shown to be useful in data translation, can be used to solve heterogeneity problems in general. Several forms of heterogeneity, such as interconnection of different operating system command languages, do not fit the model of static data objects that is used for data translation. However, it seems reasonable to suspect that whenever heterogeneity problems arise, it is possible (although perhaps not easy) to construct a conceptual model, free of implementation and user interface detail, that describes the desired functionality. If so, then a system could be developed, and a user vocabulary identified, that would solve the heterogeneity problem by giving the users access to the desired functionality without forcing them to use unfamiliar terminology or syntax.
[Bono 85] Bono, P.R.
A Survey of Graphics Standards and Their Role in Information
Interchange.

The Psychology of Human-Computer Interaction.

[Carroll 87] Carroll, John and Jean McKendree.

[Draper 85] Draper, S. W. and D. A. Norman.
Software Engineering for User Interfaces.
IEEE Transactions on Software Engineering SE-11(3):252-258,

[Farrow 82] Farrow, Rodney.
LINGUIST-86, Yet Another Translator Writing System Based on
Attribute Grammars.
In Proceedings of the SIGPLAN Symposium on Compiler

[Foley 83] Foley, J.D. and A. VanDam.
Addison-Wesley, 1983.

[Fry 81] Fry, James P. et al.
Conversion Technology, An Assessment.

The Vocabulary Problem in Human-System Communication.
Functionality and Usability.

Designing for Usability: Key principles and what designers think.

[Halstead 77] Halstead, M.
Elements of Software Science.

[Haraw 80] Haraw, P.R. and D.R. Lenorovitz.

On Designing for Usability: An Application of Four Key Principles.

[Hix 86] Hix, Deborah.
Assessment of an Interactive Environment for Developing Human-Computer Interfaces.

[Hopcroft 79] Hopcroft, J.E. and Ullman, J.D.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, 1979.

[Horak 85] Horak, W.

[Jamar 86] Jamar, Pamela G.

[Kaelbling 87] Kaelbling, Michael J.
Braced Languages and a Model of Translation for Context-Free Strings: Theory and Practice.
Man-machine interface issues in the construction and use of an expert system.  

[Knuth 68] Knuth, Donald E.  
Semantics of Context-Free Languages.  

[Lindquist 85] Lindquist, Timothy.  
Assessing the Usability of Human-Computer Interfaces.  

[Lundeberg 83] Lundeberg, M.  

A Formal Interface Design Methodology Based on User Knowledge.  

A Software Architecture for Supporting the Exchange of Electronic Manuscripts.  

Translating between Scribe, LaTeX and Troff Manuscripts.  
in preparation for submission.

[Newman 68] Newman, W.M.  
A System for Interactive Graphical Programming.  

[Parnas 72] Parnas, D. L.  
On the Criteria To Be Used in Decomposing Systems into Modules.  

[Quinn 86] Quinn, Lisa and Daniel Russell.  
Intelligent Interfaces: User Models and Planners.  
[Reisner 81] Reisner, P.

Client-Centered Design: Concepts and Experience.
*Tutorial on Software Design Techniques.*

The X Window System.

[SGML 86] Information processing - Text and office systems - Standard Generalized Markup Language (SGML)
ISO 8879-1986 (E).

[Shneiderman 87] Shneiderman, Ben.
*Designing the User Interface.*

EXPRESS: A Data EXtraction, Processing, and REStructuring System.

[Singh 86] Singh, Gurminder and Mark Green.
Automatic Generation of Graphical User Interfaces.

Developing Interactive Information Systems with the User Software Engineering Methodology.

[Yellin 86] Yellin, D.M. and Mueckstein, E.M.
The Automatic Inversion of Attribute Grammars.

*Attribute Grammar Inversion and Source-to-source Translation.*
[Yunten 86] Yunten, Tamer and H. Rex Hartson.
A SUPERvisory Methodology And Notation (SUPERMAN) for
Human-Computer System Development.
Advances in Human-Computer Interaction.

[Zave 86] Zave, Pamela and William Schell.
Salient Features of an Executable Specification Language and Its
Environment.
IEEE Transactions on Software Engineering SE-12(2):312-325,
February, 1986.
Appendix A
Syntax of MDL

The following description of the syntax of MDL is adapted from the Yacc file actually used in the implementation. Symbols consisting of all uppercase letters are terminal symbols in the MDL language.

```plaintext
input : opt_DECLARATIONS keyword_star specials
       syn_set_star opt_whitespace first_sep
       opt_whitespace prod_plus opt_whitespace
       second_sep opt_whitespace nsf_prod_star
       opt_endmarker

opt_DECLARATIONS: /* null */
    | opt_whitespace DECLARATIONS whitespace

opt_endmarker : ENDMARKER
    | /* null */

keyword_star : keyword keyword_star
    | /* null */

keyword : KEYWORD whitespace QSTRING whitespace

specials : special specials
    | /* null */

special : sf_special
    | nsf_special

sf_special : STANDARD_FORM whitespace QSTRING
           whitespace IS whitespace QSTRING
           whitespace
```
nsf_special : NON_STANDARD_FORM whitespace QSTRING
    whitespace IS nsf_designated

nsf_designated : whitespace QSTRING whitespace
    | whitespace SPECIAL whitespace IN
    whitespace STRING whitespace

whitespace : crtabsp whitespace
    | crtabsp

crtabsp : NEWLINE
    | TABS
    | BLANKS

opt_whitespace : whitespace
    | /* null */

prod_plus : prod prod_plus
    | prod

lhs : STRING

prod : lhs opt_whitespace CCE whitespace
    rhs_mdl_plus SEMICOLON whitespace

rhs_mdl_plus : rhs_mdl OR whitespace rhs_mdl_plus
    | rhs_mdl

rhs_mdl : rhs mdl_star opt_whitespace

rhs : symbol_star

symbol_star : symbol whitespace symbol_star
    | /* null */

symbol : STRING
    | QSTRING

mdl_star : mdl opt_whitespace mdl_star
    | /* null */

opt_atname : DOT STRING_EQUALS
    | /* null */

fm_func : PASS
    | WARNING
| \textsc{ABORT}  
| \textsc{IGNORE}  

\texttt{mdl} : \texttt{semantic\_func}  
| \texttt{fm\_func opt\_expr \textsc{SEMICOLON}}  
| \texttt{env\_cmd}  
| \texttt{code}  

\texttt{env\_cmd} : \texttt{BEGIN\_ENV} \texttt{whitespace context whitespace}  
| \texttt{END\_ENV} \texttt{whitespace context whitespace}  

\texttt{context} : \texttt{STRING}  

\texttt{code} : \texttt{\textsc{OBRACK} bal\_text \textsc{CBRACK} opt\_whitespace}  

\texttt{bal\_text} : \texttt{bal\_text\_part bal\_text}  
| \texttt{bal\_text\_part}  

\texttt{bal\_text\_part} : \texttt{STRING}  
| \texttt{\textsc{QSTRING}}  
| \texttt{\textsc{OBRACK} bal\_text \textsc{CBRACK}}  
| \texttt{whitespace}  
| \texttt{\textsc{SEMICOLON}}  
| \texttt{\textsc{OPAREN}}  
| \texttt{\textsc{CPAREN}}  

\texttt{opt\_expr} : \texttt{expr}  
| /* null */  

\texttt{semantic\_func} : \texttt{opt\_atname expr}  

\texttt{func} : \texttt{STRING}  
| /* null */  

\texttt{expr} : \texttt{func \textsc{OPAREN} beta\_or\_x\_plus \textsc{CPAREN}}  

\texttt{beta\_or\_x\_plus} : \texttt{beta\_or\_x \textsc{COMMA} beta\_or\_x\_plus}  
| \texttt{beta\_or\_x}  

\texttt{beta\_or\_x} : \texttt{beta}  
| \texttt{x}  

\texttt{x} : \texttt{STRING atr}  

\texttt{atr} : \texttt{\textsc{DOT} STRING}  
| /* null */
nsf_prod_star : prod nsf_prod_star
| /* null */

beta : QSTRING

first_sep : ENDMARKER
| STANDARD_FORM

second_sep : ENDMARKER
| NON_STANDARD_FORM

syn_set_star : syn_set syn_set_star
| /* null */

syn_set : CLASS whitespace STRING whitespace
syn_set_def

syn_set_def : nsfg_frag whitespace tu_means whitespace
| tu_means whitespace

tu_means : IS whitespace IGNORED
| IS whitespace BRACKETED
| MEANS whitespace QSTRING
| IS whitespace QSTRING

nsfg_frag : regexp

regexp : OPAREN regexp CPAREN
| regexp OR regexp
| regexp STAR
| regexp PLUS
| regexp regexp
| QSTRING
| STRING
Appendix B
An MDL Specification

/* Points to ponder: */
/* the BIB tag is hacked up because the bibliography */
/* is kept in another file to the refer program */
/* changed IN to INT for integrals because of */
/* conflict with MDL keyword */
/* need a special version of lex with ACOMPUTEMAX */
/* 500 as opposed to hard-coded 300 in sub2.c */
/* made definitionterm and defdeschead optional */
/* hacked mathdisplay productions in order to use */
/* math environment */
/* cdata that begins with a period or a blank should */
/* be prevented from appearing in column 1 of the doc. */
/* started to use TOPI to set off theorems and lemmas */
/* changed phrasemodelstar to paragraphtextstar in LH */
/* changed definition of cdata to be strings of print- */
/* able characters separated by whitespace. this */
/* affects phrasemodel and related things */

DECLARATIONS
KEYWORD "." /* use this to suppress break function */
KEYWORD "/" /* use this to suppress break function */
STANDARD FORM "&lt;" IS "<"
STANDARD FORM "&gt;" IS ">
STANDARD FORM "&cup;" IS " size 12 union "
STANDARD FORM "&cap;" IS " size 12 inter "
STANDARD FORM "&ne;" IS " != "
STANDARD FORM "&agr;" IS " alpha "
STANDARD FORM "&egr;" IS " epsilon "
STANDARD FORM "&empty;" IS "\(es"
STANDARD FORM "&verbar;" IS " \(or "
STANDARD FORM "&isin;" IS " \(mo "
STANDARD FORM "{" IS "{"'
STANDARD FORM "}" IS "}"')
STANDARD FORM "\n\n" IS "\n"
NONSTANDARD FORM "\*\*\*" IS 
CLASS roff_footnote "\*\*\*" IS IGNORED
CLASS roff_month "\*(MO" IS BRACKETED
CLASS roff_date "\*(DY" IS BRACKETED
CLASS start_doc MEANS"
CLASS bolditalic ".B\n.I" MEANS ".B\n.I"
CLASS bold MEANS ".B"
CLASS italic ".I" MEANS ".I"
CLASS canonical_title MEANS ".RP\n.TL"
CLASS stuff_to_ignore IS IGNORED
CLASS display_mode MEANS "I"
CLASS equation_mode MEANS "I"
CLASS roff_para IS ".LP"
CLASS roff_sect IS ".NH"
CLASS font_stuff1 ".nr PS 12" IS ".nr PS 12"
CLASS font_stuff2 ".LP" IS ".LP"
CLASS eqn_init1 ".EQ" IS ".EQ"
CLASS eqn_init2 "gsize 12" IS "gsize 12"
CLASS eqn_init3 "delim $$" IS "delim $$"
CLASS eqn_init4 ".EN" IS ".EN"
CLASS listhead_indent IS " 0 "
STANDARD FORM
article : ARTICLE frontmatter bodymatter appendixmatter
backmatter ARTICLEz
(frontmatter,font_stuff1,\nfont_stuff2,\n,eqn_init1,\n, eqn_init2,\n,eqn_init3,\n,eqn_init4,\n, frontmatter,bodymatter,appendixmatter,backmatter)
;
/* ******************frontmatter elements*************** */

frontmatter : FM titlegroup authorstar
   publisherfrontmatter fmcaptionedparagraphseqstar FMz
;
titlegroup : TIG articletitle subtitlestar TIGz
   (canonical_title,\n,articletitle,subtitlestar)
;
articletitle : ATL phrasemodelstar ATLz
   (phrasemodelstar,\n)
;
subtitlestar : subtitle subtitlestar
   |  /* optional */
;
subtitle : SBT phrasemodelstar SBTz
   (phrasemodelstar,\n)
authorstar : author authorstar
    /* optional */
    ;
author : AU namemodel AUz
    ("AU", "\n", namemodel)
    | CAU organizationnamemodel CAUz
    ("AI", "\n", organizationnamemodel)
    ;
publisherfrontmatter : PUBFM publisherfmelementstar PUBFMz
    ;
publisherfmelementstar : publisherfmelement
    publisherfmelementstar
        /* optional */
    ;
publisherfmelement : contractsponsor
    contractnumber
    | copyrightnotice
    | reprintsource
    | distributor
    | acidfreepaper
    | price
    | extentofwork
    | coden
    | ordernumber
    | articleidentifier
    | issn
    ;
contractsponsor : CGS organizationnamemodel CGSz
    ("Contract Sponsor:", organizationnamemodel, "\n")
    ;
contractnumber : CGN phrasemodelstar CGNz
    ("Contract number:", phrasemodelstar, "\n")
    ;
copyrightnotice : CRT copyrightdataplus CRTz
    ;
copyrightdataplus : copyrightdata copyrightdataplus
    copyrightdata
    ;
copyrightdata : copyrightdate
    copyrightname
    | clearancecenter
    ;
copyrightdate : CRD datemodel CRDz
    ("Copyright ", datemodel)
    ;
copyrightname: CRN

CRN organizationname

("Copyright name: ", organizationname, ";")

clearancecenter: CCI

CCI organizationname

("Clearance center: ", organizationname, ";")

reprintsource: RPS

RPS organizationname

("Reprint source: ", organizationname, ";")

distributor: AVL

AVL organizationname

("Distributor: ", organizationname, ";")

acidfreepaper: PHI

PHI phrasemodelstar

("Acid free paper: ", phrasemodelstar, ";")

price: PRC

PRC phrasemodelstar

("Price: ", phrasemodelstar, ";")

extentofwork: EXT

EXT phrasemodelstar

("Extent of work: ", phrasemodelstar, ";")

coden: CDN

CDN phrasemodelstar

("Coden: ", phrasemodelstar, ";")

ordernumber: AON

AON phrasemodelstar

("Order number: ", phrasemodelstar, ";")

articleidentifier: AID

AID phrasemodelstar

("Article identifier: ", phrasemodelstar, ";")

issn: ISSN

ISSN phrasemodelstar

("ISSN: ", phrasemodelstar, ";")

fmcaptionedparagraphseq: fmcaptionedparagraphseq

fmcaptionedparagraphseq: dedication

abstract

supportingmaterial

dedication: DED

DED captionedparagraphseq

("Dedication: ", captionedparagraphseq)

abstract: ABS

ABS captionedparagraphseq
roman
| smallcaps
| emphasis1
| emphasis2
| emphasis3

italics : IT phrasemodelstar ITz
("I","\n",phrasemodelstar,".R","\n")
;

bold : B phrasemodelstar Bz
("B","\n",phrasemodelstar,".R","\n")
;

bolditalics : BI phrasemodelstar BIZ
(bolditalic,"\n",phrasemodelstar,".R","\n")
;

roman : RM phrasemodelstar RMz
("roman: ",phrasemodelstar)
;

smallcaps : SCP phrasemodelstar SCPz
("smallcaps: ",phrasemodelstar)
;

emphasis1 : EI phrasemodelstar Elz
("\n",".I","\n",phrasemodelstar,"\n",".R","\n")
;

emphasis2 : E2 phrasemodelstar E2z
("\n",".nf","\n",phrasemodelstar,"\n",".fi","\n")
;

emphasis3 : E3 phrasemodelstar E3z /* display_mode */
(".DS ",display_mode,"\n",phrasemodelstar,
".DE","\n")
;

references : NTR phrasemodelstar NTRz
("**","\n",".FS **","\n",phrasemodelstar,"\n"," .FE"","\n")
| FNR phrasemodelstar FNRz
("\n","[","\n",phrasemodelstar,"\n","]","\n")
| FGR phrasemodelstar FGRz
("Figure ",phrasemodelstar)
| TBR phrasemodelstar TBRz
("Table ",phrasemodelstar)
| ARTR phrasemodelstar ARTRz
("Plate ",phrasemodelstar)
| APR phrasemodelstar APRz
("APR ",phrasemodelstar)
| SRR phrasemodelstar SRRz
("SRR ",phrasemodelstar)
|        RB phrasemodelstar RBz
|       "RB ",phrasemodelstar)
|       ;
|/* roff supports one letter only */
greek : GR phrasemodelstar GRz
|       "\(*",phrasemodelstar)
|       ;
cyrillic : CYR phrasemodelstar CYRz
|       "Cyrillic: ",phrasemodelstar)
|       ;
embeddedquote : EMQ phrasemodelstar EMQz
|       "\QF","\n",phrasemodelstar)
|       ;
|/* ***************paragraphsubelements*************** */
paragraphsubelements : generalelement
|       |
|       |   tablematter
|       |   list
|       |   mathformula
|       ;
mathformula : displayformula /* equation_mode */
|       ("\n",".EQ ",equation_mode,"\n",displayformula,
|       "\n",".EN","\n")
|       | inlineformula
|       ("\$",inlineformula,"\$")
|       ;
displayformula : FD labeledformulaplus FDz
|       ;
labeledformulaplus : labeledformulaplus labeledformula
|       (labeledformulaplus,"\n",labeledformula)
|       | labeledformula
|       ;
labeledformula : label label formulaline
|       ;
label : LA charorhorizstar LAz
|       /* optional */
|       ;
formulaline : FL builtuptextstar FLz
|       (builtuptextstar,"\n")
|       ;
charorhorizstar : charorhorizstar charorhoriz
|       /* optional */
|       ;
charorhoriz : charstringtext
|       horizontalelement
|       ;
inlineformula : F builtuptextstar Fz
/* **Complex built-up constructs group - f-bujsx**** */
builtuptext*star : builtuptext*star builtuptext
  |  /* optional */
;
builtuptext : builtupelement
  |  speciallimit
  |  charstringtext
  |  specialcharstring
  |  horizontalelement
;
/* *****Built-up elements group - f-bu**** */
builtupelement : fraction
  |  limit
  |  radical
  |  array
  |  fence
;
  fraction : FR numerator denominator FRz
;
  numerator : NU builtuptext*star NUz
;
  denominator : DE builtuptext*star DEz
;
  limit : LIM operator lowerlimit upperlimit operand LIMz
;
  operator : OP charstringtext OPz
  |  OP romanfunction OPz
  |  OP horizontalelement OPz
;
  lowerlimit : LL builtuptext*star LLz
;
  upperlimit : UL builtuptext*star ULz
;
  operand : OPD builtuptext*star OPDz
  |  /* optional */
;
  radical : RAD radicand radix RADz
;
  radicand : RCD builtuptext*star RCDz
;
  radix : RDX builtuptext*star RDXz
  |  /* optional */
;
  array : AR arrayrowplus ARz
;
arrayrowplus : arrayrowplus arrayrow
  | arrayrow
  ;
arrayrow : ARR arraycellplus ARRz
  ;
arraycellplus : arraycellplus arraycell
  | arraycell
  ;
arraycell : ARC builtuptextstar ARCz
  ;
fence : FEN builtuptextstar cpostandtextstar rightpost
  FENz
  ;
cpostandtextstar : cpostandtextstar cpostandtext
  | /* optional */
  ;
cpostandtext : centerpost builtuptextstar
  ;
centerpost : CP CPz
  ;
rightpost : RP RPz
  ;
speciallimit : product
  | integral
  | summation
  ;
product : PR lowerlimit upperlimit operand PRz
  ;
integral : INT lowerlimit upperlimit operand INTz
  ;
summation : SUM lowerlimit upperlimit operand SUMz
  ;
/* ******Character string text group - f-cstxt***** */
charstringtext : /* cdata */ phrasemodel
  | charstring
  ;
/* ******Special characters/strings group - f-scs***** */
specialcharstring : romanfunction
  | increment
  | vector
  | dyadic
  | field
  ;
romanfunction : RF RFz
  ;
increment : INC INCz
vector : V Vz

dyadic : DY DYZ

field : FI FIZ

/* *****Character string group - f-cs***** */
/* emphasis used to be in this list, but using */
/* phrasemodel instead of cdata in charstringtext */
/* makes this redundant and causes many conflicts */
charstring : accentelement
  | typestyle
  | mathgreek
  | boldgreek
  | spacepositiontag

accentelement : A accentcomponent accentcomponent Az

accentcomponent : AC charorspecstar ACz

charorspecstar : charorspecstar charorspec
  | /* optional */

charorspec : charstringtext
  | specialcharstring

mathgreek : G cdata Gz
  ("math greek:" , cdata)

boldgreek : BG cdata BGz
  ("bold greek:" , cdata)

/* *****p.fnt.ph***** */
typestyle : script
  | german
  | openface
  | sansserif
  | monospace
  | boldscript
  | boldgerman
  | bolditalicsansserif
  | boldsansserif
  | italicsansserif

script : SC charstringtextstar SCz
("Script: ",charstringtextstar)
;
german : GE charstringtextstar GEz
("German: ",charstringtextstar)
;
openface : OP charstringtextstar OPz
("Openface: ",charstringtextstar)
;
sansserif : SSF charstringtextstar SSFz
("Sansserif: ",charstringtextstar)
;
monospace : TY charstringtextstar TYz
("Monospace: ",charstringtextstar)
;
boldscript : BSC charstringtextstar BSCz
("Boldscript: ",charstringtextstar)
;
boldgerman : BGE charstringtextstar BGEz
("Bold German: ",charstringtextstar)
;
bolditalicsansserif : BISF charstringtextstar BISFz
("Bold Italic Sansserif: ",charstringtextstar)
;
boldsansserif : BSF charstringtextstar BSFz
("Bold Sansserif: ",charstringtextstar)
;
italicsansserif : ISF charstringtextstar ISFz
("Italic Sansserif: ",charstringtextstar)
;
charstringtextstar : charstringtextstar charstringtext
/* optional */
;
;/* *****sp.pos***** */
spacepositiontag : horizontalmark
|
  horizmarkref
|
  verticalspace
|
  horizontalspace
|
  turnline
|
  zerowidth
|
  verticalmark
|
  vertmarkref
;

horizontalmark : HMK HMKz
;

horizmarkref : HMKR HMKRz
;
verticalmark : VMK VMKz
;
vertmarkref : VMKR VMKRz
;
horizontalspace : HSP HSPz
;
verticalspace : VSP VSPz
;
turnline : TU TUz
;
zerowidth : ZW ZWz
;
/* **Horizontally oriented elements group - f-ph***** */
horizontalelement : boxedelement
|    underline
|    overline
|    superior
|    inferior
;
boxedelement : BOX builtuptextstar BOXz
;
underline : UNL builtuptextstar UNLz
;
overline : OVL builtuptextstar OVLz
;
superior : SUP builtuptextstar SUPz
( " sup ",builtuptextstar," " )
;
inferior : INF builtuptextstar INFz
( " sub ",builtuptextstar," " )
;
/* *****Nonmath text group***** */
nonmathtextstar : nonmathtextstar nonmathtext
|    /* optional */
;
nonmathtext : paragraph
|    emphasis
|    list
|    references
|    inlinequote
|    cdata
;
/* *****things that cannot be gotten to**** */
stack : STK layerplus STKz
;
layerplus : layerplus layer
|      layer
;
layer : LXR builtuptextstar LRYz
;
atomchange : ACH builtuptextstar ACHz
;
phrase : PHR nonmathtextstar PHRz
;
/* *** End of things that cannot be gotten to *** */
generalelement : definitionlist
|  organizationaddress
|  individualaddress
|  artwork
|  blockquote
|  literaltext
|  publicationdate
|  bibrefintext
|  biblist
|  itemlist
|  author
;
definitionlist : DL definitionitemstar DLz
;
definitionitemstar : definitionitem definitionitemstar
|  /* optional */
;
definitionitem : definitionheads definitionterm
  definitiondesc
;
definitionheads : deftermhead defdeschead
  ("\n",".B",deftermhead,\"\n",defdeschead)
|  /* optional */
  ("\n",".B Definition","\n")
;
deftermhead : DTHD phrasemodelstar DTHDz
;
/* defdeschead used to be optional */
defdeschead : DDHD phrasemodelstar DDHDz
;
definitionterm : DT phrasemodelstar DTz
  (phrasemodelstar,"\n")
|  /* optional */
;
definitiondesc : DD paragraphseq DDz
;
organizationaddress : OAD organizationnamemodel OADz

individualaddress : IAD namemodel IADz

artwork : ART /***********/ ARTz

blockquote : BQ paragraphseq BQz

literaltext : LIT phrasemodelstar LITz
(".nf\n", phrasemodelstar, "\n", ".fi\n")

bibrefintext : BB bibentry BBz

biblist : BIBL bibitemplus BIBLz

bibitemplus : bibitemplus bibitem
| bibitem

bibitem : listhead bibrefintext

itemlist : ITML itemlistitemplus ITMLz

itemlistitemplus : itemlistitemplus itemlistitem
| itemlistitem

itemlistitem : listhead item

item : ITM subcomponent1 subcomponent2star pagenumrefstar ITMz

subcomponent1 : SIT1 phrasemodelstar SIT1z

subcomponent2star : subcomponent2star subcomponent2list
| /* optional */

subcomponent2list : subcomponent2 subcomponent3star

subcomponent2 : SIT2 phrasemodelstar SIT2z

subcomponent3star : subcomponent3 subcomponent3star
| /* optional */

subcomponent3 : SIT3 phrasemodelstar SIT3z

tablematter : TBL number tabletitle tablebody tablesoure TBLz
listheaditemstar : listheaditemstar listheaditem
| /* optional */
;
listheaditem : listhead listitem
("\n",'IP ",listhead," ',listhead_indent,"\n",
listitem)
;
listhead : LH /* phrasemodelstar */ paragraphtextstar LHz
| /* optional */
;
listitem : LI paragraphseq LIZ
;
/* **********headings********** */
headings : H phrasemodelstar Hz
("\n",phrasemodelstar)
| H1 phrasemodelstar H1z
("H1\n",phrasemodelstar)
| H2 phrasemodelstar H2z
("H2\n",phrasemodelstar)
| H3 phrasemodelstar H3z
("H3\n",phrasemodelstar)
| H4 phrasemodelstar H4z
("H4\n",phrasemodelstar)
;
/* **********topics********** */
topics : TOP1 captionedparagraphseq TOP1z
("\n",".br","\n \n",captionedparagraphseq,"\n")
| TOP2 captionedparagraphseq TOP2z
("Topic 2\n",captionedparagraphseq)
| TOP3 captionedparagraphseq TOP3z
("Topic 3\n",captionedparagraphseq)
| TOP4 captionedparagraphseq TOP4z
("Topic 4\n",captionedparagraphseq)
;
/* **********appendix matter********** */
appendixmatter : APPM appendixplus APPMz
| /* optional */
;
appendixplus : appendixplus appendix
| appendix
;
appendix : number appendixtitle sectionsubelement star
    section star

appendixtitle : AFT phrasemodel star APTz
    ("Appendix ", phrasemodel star)

section star : section star section
    /* optional */

/* ****************** back matter ********************* */
backmatter : BM bmcaptionedparagraphseq star BMz
    (bmcaptionedparagraphseq star)
    /* optional */

bmcaptionedparagraphseq star : bmcaptionedparagraphseq star
    bmcaptionedparagraphseq
    /* optional */

bmcaptionedparagraphseq : acknowledgement
    bibliography
    vita

acknowledgement : ACK captionedparagraphseq ACKz
    ("\n", \LF", 
", captionedparagraphseq)

bibliography : BIB captionedparagraphseq BIBz
    (".bp", 
", captionedparagraphseq)
    BIB  /* since refer handles this */
    (".bp", 
", ".", 
", 
", "$LIST$", 
", 
", ".")

vita : VT captionedparagraphseq VTz

/* **************** models ****************************** */
/* **************** address *************************** */
addressmodel : addresselement star

addresselement star : addresselement addresselement star
    /* optional */

addresselement : street
    city
    countrysubdivision
    country
postalcode
standardaddressnumber
electronicaddress

; street : STR phrasemodelplus STRz
   (phrasemodelplus, "\n")
;
; city : CTY phrasemodelplus CTYz
   (phrasemodelplus, ", ")
;
; countrysubdivision : SBD phrasemodelplus SBDz
   (phrasemodelplus, "\n")
;
; country : CNY phrasemodelplus CNYz
   (phrasemodelplus, "\n")
;
; postalcode : PC phrasemodelplus PCz
   (phrasemodelplus, "\n")
;
; standardaddressnumber : SAN phrasemodelplus SANz
;
; electronicaddress : EAD phrasemodelplus EADz
   (phrasemodelplus, "\n")
;
/* ************************************name******************************* */
name: firstname surname degreeorschoolstar rolestar
   addresselementstar affiliation

;
; firstname : FNM phrasemodelstar FNMz
   (phrasemodelstar)
   |  /* optional */
;
; surname : SNM phrasemodelstar SNMz
   (phrasemodelstar, "\n")
;
; degreeorschoolstar : degreeorschoolstar degreeorschool
   |  /* optional */
;
; degreeorschool : DEG phrasemodelstar DEGz
   |   SCH organizationnamemodel SCHz
;
; rolestar : rolestar role
   |  /* optional */
;
; role : ROLE phrasemodelplus ROLEz
;
affiliation : AFF organizationnameorganizationnameorganizationname

| /* optional */

("\n")

;

/****organization name************ */

organizationnameorganizationnameorganizationname

organizationdivisionorganizationdivisionorganizationdivision

| /* optional */

;

organizationdivision : ODV organizationdivisionorganizationdivisionorganizationdivision

/****date************ */

dateorganizationdateorganizationdateorganizationdate

monthorganizationmonthorganizationmonthorganizationmonth

| /* optional */

;

dayorganizationdayorganizationdayorganizationday

| /* optional */

;

yearorganizationyearorganizationyearorganizationyear

;

/****float elements************ */

floatorganizationfloatorganizationfloatorganizationfloat

figureorganizationfigureorganizationfigureorganizationfigure

| footnoteorganizationfootnotefootnotefootnote

| noteorganizationnoteorganizationnoteorganizationnote

figure : FIG /************/ FIG

;

foot_number : number

;

footnote : FN foot_numberorganizationfoot_numberorganizationfoot_numberorganizationfoot_number

(".FS ",foot_number,"\n",organizationfoot_number,"\.FE","\n")

;
noteintext : bibentry

bibentry : number title bibelementstar

title : seriestitle
  | articletitle
  | sectiontitleinserial

bibelementstar : bibelementstar bibelement
  | /* optional */

bibelement : author
  | monographseriesnum
  | seriestitle
  | publisherlocation
  | publicationdate
  | pagenumref
  | articletitle
  | sectiontitleinserial
  | otherbibinfo

monographseriesnum : MSN cdata MSNz
  ("Monograph series number: ", cdata)

seriestitle : SRT phrasemodelstar SRTz
  ("Series title: ", phrasemodelstar)

publisherlocation : LOC addressmodel LOCz

publicationdate : PDT datemodel PDTz
  ("Published: ", datemodel)

sectiontitleinserial : SCT phrasemodelstar SCTz
  ("Section title: ", phrasemodelstar)

otherbibinfo : OBI phrasemodelstar OBIz

/* ****captioned paragraph sequence **** */
captionedparagraphseq : opt_heading paragraph sectionsubelementstar

opt_heading : heading
  | /* null */

heading : H phrasemodelstar Hz
  (".B","\n", phrasemodelstar,"\n",".R","\n")
; /* ********phrase model********** */
phrase model star : phrase model plus 
| /* optional */
;
phrase model plus : phrase model phrase model star 
;
phrase model : cdata
| phrases 
;
/* *****paragraph sequence***** */
paragraph seq : /* Charles removed paragraph */
paragraph or subelement star
| /* this used to be optional */
;
paragraph or subelement star : paragraph or subelement 
paragraph or subelement star
| /* optional */
;
paragraph or subelement : paragraph
| paragraph subelements 
; 
%%
start_doc : ".RP" newline
| /* null */
;
new line : "\n"
;
whitespace : whitespace char whitespace 
| whitespace char 
;
whitespace char : newline
| "\t"
| " "
;
/* null, since roff allows frontmatter to be omitted */
frontmatter :
;
/* null, since footnote numbers may be omitted */
foot number : text
display_mode : "I"
   | "L"
   | "C"
   | "B"
;

equation_mode : "I"
   | "L"
   | "C"
;

roff_para : ".LP"
   | ".PP"
;

roff_sect : ".NH"
   | ".SH"
;

canonical_title : ".tl" text "/" text "/" text newline
   | ".TL" newline
;

w_l : cdata /* used to be whitespace letter */
;

w_l_plus : w_l w_l_plus
   | w_l
;

w_t : whitespace text
;

w_t_plus : w_t w_t_plus
   | w_t
;

center : ".ce" w_t
   | ".ce"
;

bold : ".bd" w_l w_l
   | ".bd" w_l w_l w_l
;
end_para: 
  
listhead_indent : cdata

indent : 
  
underline : 
  
stuff_to_ignore : stuff_to_ignore1

stuff_to_ignore1
  
stuff_toIgnore2

stuff_to_ignore2
  

".mk"
".na"
".ne" w_t
".ne"
".nf"
".nh"
".nm" w_t w_l w_l w_l
".nm"
".nn" w_t
".nn"
".nr" w_l w_t w_t
".ns"
".nx" whitespace text
    {fprintf(stderr,".nx is not supported.\n");}
    &ERROR;
".os"
".pc" w_l
".pi" w_t
".pm"
".pm t"
".ps" w_t
".ps +" text
".ps -" text

stuff_to_ignore2
  : ".pl" w_t
    ".pl"
    ".pn" w_t
    ".pn"
    ".po" w_t
    ".po"
    ".rn" w_t w_t
    ".rn"
    ".rm" w_t
    ".rm" "."
    ".rr" w_l
    ".rs"
    ".rt" w_t
    ".rt"
    ".ss" w_t
    ".ss"
    ".sv" w_t
    ".sv"
"ta" w_t_plus
"tc" w_l
"tc"
"tm" w_t
"tr" w_t
"uf" w_t
"uf"
"vs" w_t
"vs"
"wh" w_t w_t
"1C"
"2C"
"AM"
"AT"
"BT"
"BT" w_t
"CM"
"DA"
"DA" w_t_plus
"EF" whitespace "" text "" text "" text ""
"EN"
"EQ" w_t
"EQ I" w_t
"EQ C" w_t
"EQ L" w_t
"IX" w_t_plus
"KE"
"KF"
"KS"
"ND" w_t
"ND"
"OF" w_t
"OF"
"OH" w_t
"OH"
"PT"
"PT" w_t
"PX" w_l
"PX"
"RE"
"RP"
"RS"
"TA" w_t_plus
"XA" w_t w_t
"XE"
"XS" w_t w_t
bad_news : ".rd" w_t

{ fprintf(stderr,".rd is not supported.\n") ;
 &ERROR;
 |".so" w_t

{ fprintf(stderr,".so is not supported.\n") ;
 &ERROR;
 | ".CT"

{ fprintf(stderr,"Thesis mode is not supported.\n") ;
 &ERROR;
 |".TM"

{ fprintf(stderr,"Thesis mode is not supported.\n") ;
 &ERROR;
 | ".pl"

{ fprintf(stderr,"Thesis mode is not supported.\n") ;
 &ERROR;
};