DEXTER: Generating Documents by means of computational registers

Joseph D. Oldham

University of Kentucky, oldham@cs.uky.edu
Chapter 1

Introduction

I have designed and implemented an approach to generating human-readable electronic documents in a variety of domains. The generation process accepts structured data as input and outputs a document that presents the data values and satisfies user expectations. For example, here is part of an output document:

Victor March is an old, slightly overweight male. His wife is Elizabeth. ...

The example text conveys the following information about Victor March: His age is between 51 and 70 years, his body mass index, derived from his height and weight, is somewhere in the range 24 to 27, he is married, and his wife’s name is Elizabeth March. The expression is reasonably fluent English. It is not simply a set of database tables representing that Victor March is 56 years old, 69 inches tall and 178 pounds and so on.

Electronic documents are documents stored online in some representation, such as HTML or RTF [13, 65]. Such documents present data to a user and may include multimedia. My software accepts data as input and produces an electronic document as output. The generation is systematic because it proceeds through the same logical steps regardless of the domain of the data or the eventual form of the document. The input data is structured. It has a known storage format (such as a relational database) and a known logical scheme within that storage format. User expectations are the conventions a document must follow and the constraints a document must satisfy. Examples are the use of the appropriate
vocabulary, specific ordering of information, and applying general terms such as “slightly overweight” (as above) in a correct manner. The document generator is called a computational register (“register” for short) and is produced by register-authoring software. The idea is represented in Figure 1.1.

![Diagram](image)

Figure 1.1: Computational registers

### 1.1 Motivation

Computers store much of our data. A Data Storage System (DSS) on a computer may be anything from a collection of plain text files to a many-featured database management system (DBMS) with powerful query facilities. An individual granted access to a DSS is called a user of the system. To use a system’s data the user must be able to query the system and understand the results. The query and retrieval facilities of electronic DSSs can be deficient both in the queries allowed and the manner in which results are displayed.

DSSs store data under some scheme — a way of representing and organizing data. DSSs do not assign meaning to the data. Users want information: data made useful by being placed in a context based on some meaning the user assigns the data. The user faces problems in getting the DSS to yield information. Problems include (1) understanding how to use the software necessary to view the data, (2) understanding the data scheme, (3) formulating the queries required to get the right data, and (4) sifting through irrelevant results. Sometimes one composite unit of information requires multiple data queries. When users are seeking aberrations in data values they must sometimes wade through streams of predicted, expected values that are not interesting. A user seeks data with a purpose in
mind, and it can be very difficult to make a given DSS cooperate in meeting the user goal. DSS users cannot become data users without first resolving problems of the kind I have just mentioned. There is often an impedance mismatch between users and DSS; the transformation of system data to user information is inefficient. The challenge posed by modern, large DSSs is to help users more efficiently achieve their own purpose. Friendlier query and retrieval tools are needed.

The impedance mismatch between users and DSS is largely an issue of data presentation. Data is presented by unfamiliar programs in an unfamiliar representation. Too much data is presented. Some data storage systems put little effort toward presentation, simply showing a raw form of data in its storage scheme. Others, such as the Illustra\(^1\) database system, try to manage presentation of complex data through special software tools. Many have some mechanism, such as the view facilities defined for SQL, or some report definition facility such as Access’ Report Wizard, to try to control what the user sees. Generating documents enhances what the user sees [2, 17, 33]. Consider some typical DSS outputs:

Example 1.1 Registrar’s screen

<table>
<thead>
<tr>
<th>111 STUDENT TERM ATTRIBUTES</th>
<th>Beast, Will D.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCREEN:</td>
<td>TERM: 995</td>
</tr>
<tr>
<td>INST: JU</td>
<td></td>
</tr>
<tr>
<td>AU: LC</td>
<td></td>
</tr>
<tr>
<td>SID: 9999999999 CRNS:</td>
<td>ROLL SCR:</td>
</tr>
<tr>
<td>PRIMARY PROGRAM</td>
<td>MISCELLANEOUS DATA</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>CAREER: UC</td>
<td>DEAN’S LIST: N</td>
</tr>
<tr>
<td>CLASSIFICATION: UN1</td>
<td>DORM:</td>
</tr>
<tr>
<td>COLLEGE/CAT: EN 993</td>
<td>ROOM NUMBER:</td>
</tr>
<tr>
<td>EOT ACADEMIC ACTN: G</td>
<td>ROOM TYPE:</td>
</tr>
</tbody>
</table>

\(^1\)Illustra is now part of the Informix Universal Server.
This screen contains some of the most basic information in the database. To anyone accustomed to academic records this screen provides the following information:

Will D. Beast is a freshman working toward a Bachelor of Science in Computer Science in the College of Engineering. He or she is enrolled full time, is not currently on the Dean’s list, and is advised by J. Doe. The student probably does not have a minor.

The user may need to guess at the meaning of abbreviations such as **BSCS** and **EN**. The screen shows numerous empty fields, many with opaque field names or values. For example, it is not obvious that the TERM of 995 means Fall 1999. The screen presents the data indirectly for the purposes of most users, who often require a separate lookup to explain fields and values.

The next example is output from a tool called Pulsar, which has various ways to monitor computers on a network [19]. Pulsar can present its data graphically or more primitively. Example 1.2 shows a more primitive form of Pulsar presentation.
Example 1.2 Pulsar’s text output

agloval
disk:/winnt 17 Fri Sep 24 15:42:34 disk /winnt at 94% of capacity; 32136 left
al
disk:/usr 17 Fri Sep 24 13:25:06 disk /usr at 94% of capacity; 21072 left
wtmpx 15 Fri Sep 24 13:25:10 /var/adm/wtmpx is 2173968 bytes
andal
mountd 15 Sat Sep 25 06:34:25 mountd is not running
swap 15 Sat Sep 25 10:23:34 Free swap 12472
baker
ping 30 Sat Sep 25 09:57:09 Does not respond to ping from al.cs.emgr.uky.edu
pulsar 15 Sat Sep 25 07:23:17 no recent report
cashew
disk:/export 20 Fri Sep 24 13:06:27 disk /export at 96% of capacity; 62333 left
memory 15 Sat Sep 25 09:46:28 X has size 46344 (2150 resident)
golf
memory 15 Sat Sep 25 07:15:40 rpc.mountd has size 3100 (2300 resident)
hoopoe
cpu 17 Sat Sep 25 10:00:02 load is 1.08
zonker
power 15 Fri Sep 24 17:30:23 AC Power has failed, system is running on UPS power.
Fri Sep 24 13:58:59 EDT 1999

Someone who knows a reasonable amount about how computers on networks can be monitored and the network this data represents must still know Pulsar’s data scheme to understand this presentation of the data.

A register could input this presentation and generate a document including the following:

The machine baker is down.

Machines al and agloval report disks nearing capacity:

al   : /u/al_d3 has 103045 KB
      : /usr jas 21072 KB left
agloval : /winnt has 32136 KB left

Hoopoe’s cpu was busy at last check.
The meaning of the tables in Example 1.3 is probably clear to anyone who has an idea about databases, but they are an abominable description of people and their pets. It would not clarify the data to add the Kind and Name columns and discard the second table. Any use of this data will require effort on the part of the user.

**Example 1.3** Relational tables

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
<th>Age</th>
<th>Sex</th>
<th>HtIn</th>
<th>WtLb</th>
<th>Eyes</th>
<th>Hair</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natalia</td>
<td>Sturek</td>
<td>20</td>
<td>F</td>
<td>64</td>
<td>105</td>
<td>br</td>
<td>b1</td>
<td>S</td>
</tr>
<tr>
<td>Victor</td>
<td>March</td>
<td>56</td>
<td>M</td>
<td>69</td>
<td>176</td>
<td>bl</td>
<td>gr</td>
<td>M</td>
</tr>
<tr>
<td>Elizabeth</td>
<td>March</td>
<td>51</td>
<td>F</td>
<td>64</td>
<td>120</td>
<td>bl</td>
<td>b1</td>
<td>M</td>
</tr>
<tr>
<td>Joseph</td>
<td>Oldman</td>
<td>40</td>
<td>M</td>
<td>67</td>
<td>180</td>
<td>gr</td>
<td>b1</td>
<td>M</td>
</tr>
<tr>
<td>Dianna</td>
<td>Hobart</td>
<td>35</td>
<td>F</td>
<td>63</td>
<td>110</td>
<td>bl</td>
<td>b1</td>
<td>M</td>
</tr>
<tr>
<td>Miroslaw</td>
<td>Szczygelski</td>
<td>42</td>
<td>M</td>
<td>72</td>
<td>180</td>
<td>br</td>
<td>br</td>
<td>M</td>
</tr>
<tr>
<td>Helena</td>
<td>Szczygelska</td>
<td>41</td>
<td>F</td>
<td>68</td>
<td>145</td>
<td>br</td>
<td>br</td>
<td>M</td>
</tr>
<tr>
<td>Beth</td>
<td>Silverstein</td>
<td>44</td>
<td>F</td>
<td>66</td>
<td>120</td>
<td>br</td>
<td>br</td>
<td>M</td>
</tr>
<tr>
<td>Raphael</td>
<td>Winkler</td>
<td>45</td>
<td>M</td>
<td>69</td>
<td>140</td>
<td>br</td>
<td>bk</td>
<td>M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
<th>Kind</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Victor</td>
<td>March</td>
<td>cat</td>
<td>Carmen</td>
</tr>
<tr>
<td>Elizabeth</td>
<td>March</td>
<td>cat</td>
<td>Carmen</td>
</tr>
<tr>
<td>Beth</td>
<td>Silverstein</td>
<td>badger</td>
<td>Stalwart</td>
</tr>
<tr>
<td>Raphael</td>
<td>Winkler</td>
<td>badger</td>
<td>Stalwart</td>
</tr>
</tbody>
</table>

The presentation is inadequate for most users in any of these cases. There are users who are put off from using DSSs due in part to poor presentation.

The previous examples show cases where presentation via documents would be useful but might not be critical. The following final example is a scenario where presentation of a
Example 1.4 Online examination

The data must be used to support an online examination of a group of highly skilled individuals such as physicians\(^2\). We want to use simulations, in this case evolving a simulated patient, to see how the physician will respond. Our simulation will make use of our database to represent the patient’s health state.

In this case there is a very specific form of presentation that we should use. The physician examinees are familiar with presentation of patients through progress notes, history and physical reports and so on. They do not see patients presented as tables of raw data. Primitive presentation, or even unexpected presentation, would both frustrate physicians and make evaluation of test results difficult. We would face the problem of distinguishing physician response to unfamiliar presentation from physician response to data. Instead the patient presentation on the exam should include accurate representation of patient data through simulated clinic and hospital documents. Ideally we would have a transparent presentation; the physician would be able to forget it was a simulation or test. Example 1.4 describes a situation where users need a good presentation that can be delivered by correctly generated electronic documents. I am interested in software to build such documents.

Document generation requires a thought-out approach. All programs have a data-presentation component, even if it is completely trivial, such as showing a file via a system utility. There are programming languages that offer some support for presentation. Perl, for instance, implements “format”, which defines a fixed template [75]. We often build more elaborate presentation components into our software tools, but most often we build presentation components based on our needs of the moment, with the tools at hand, and with no thought toward any guiding approach. Presentation components built in this manner tend to be inadequate from the beginning and will degrade as needs change. What are the general requirements for intelligent document generation?

Good presentation in a document requires flexibility of data expression based on: values in the data; what the user wants to do with the data; and how the DSS stores the data.

\(^2\)For example as in the ABFP Computer Based Testing Project [73].
Accounting for data values and user goals are closely related concerns. Data that is “as expected and unremarkable” might be suppressed from presentation in some settings. Unexpected data should usually be prominently displayed. We cannot recognize “expected”, “unexpected”, “(un)important” or “significant” without understanding what the user will do with the data; building a presentation system accounts for the different uses of the data. (Both an insurance company auditor and a physician need to use the same patient records, but their different purposes cause them to view data differently.) Data representation also influences presentation. Different representations lend themselves to different kinds of presentation. A good approach to data presentation must account for varieties of data representation.

1.2 Problem Statement

In this thesis I address the following questions:

How do you take structured data, as in a relational database, and represent that data in an electronic document while taking into account both needs and expectations of a class of users, as well as actual values in the data? Can you accomplish the task using tools familiar to most computer programmers? Can you describe the process so that it is applicable to a variety of domains? Can you give a systematic treatment to the described process?

My problem statement contains several qualified terms: structured data, needs and expectations of a class of users, familiar tools and a systematized process. I have introduced these terms at the very beginning of this chapter, but now I will say a little more about what I mean and why I make these qualifications.

Structured data is stored in a known format and subject to queries in a well defined language. Knowing my input data is structured allows me to be sure I can formulate queries on the data at the time that I define how data is to be used in a document. Otherwise the computational register software would have to be able to understand arbitrary data on examination. Solving that problem is not within the scope of my research.

I am interested in presentation software that responds to the expectations of a class of users rather than to the expectations of individual users because it is practical. Modeling expec-
tations and preferences of an individual is a larger problem than modeling expectations of a group, where the interesting expectations are those shared by most of the users in the group. Some effort and expense is required to build a document generator. A document serving a group of users rather than an individual makes economic sense. Finally, understanding presentation to a group is an incremental step toward understanding presentation to each member of the group.

Using standard computer programming tools is important. Building software, including document generation software, requires someone with programming skills. The population of programmers we can draw on for building presentation systems is much larger if no technical knowledge beyond programming is required. For instance, we do not want to require the assistance of a formally trained computational linguist every time we want to build a document generator. Because many enterprises employ programmers, an approach to document generation that requires only programming skills and domain knowledge will be more widely applicable than an approach that requires less common kinds of knowledge.

Software is used to display data in all domains where computers are used. Therefore, an approach to document generation should be applicable in any domain where generating a document makes sense. We always try to build more general-purpose tools and tools that can be maintained as needs change. Lacking general-purpose tools, we prefer a general approach underlying our particular tools. I do not know how to approach building one tool that will create an arbitrary document. I do have an idea how to build software that will help build a number of specific document generators in a variety of domains. Compiler generators provide a good analogy. A compiler translates one language into another. We do not have an all-purpose compiler. We do have compiler generators. A compiler generator knows enough about compilers to take descriptions of compiler input and output for a specific translation and generate the needed compiler. Compiler generators are possible because there is a well defined approach to compilation. Registers are like compilers: they translate specific data into a specific document. I cannot approach a general-purpose register, but I can systematically describe the process a register uses. With that systematic description I can provide a software package, called a register system, to help design, build and maintain specific registers.
1.3 Sketch of the Approach

In subsequent chapters I look at registers at the architectural level, then at the level of design decisions, and finally at the level of implementation using DEXTER. At present I give only a brief outline of document generation by means of registers.

Figure 1.1 shows the context in which a register exists. Figure 1.2 describes registers internally. Queries are sent to the DSS, which returns results. The module that deals with queries and their results is called the field. The field deals with the data directly and puts results of queries into a form amenable to presentation given the specific needs for the current target document. It is the only module that must understand the input data structure shown. The mode examines the query results as represented by the field and determines what could be said in the document. The mode then takes what can be said and plans the structure of the document, resulting in a document outline. The outline is passed to a module called the tenor. In the tenor the outline is transformed into a document represented in some language such as HTML. All of these components require the knowledge of user expectations shown in Figure 1.1. A more detailed view is given in Figure 1.3.

I base the field and mode of registers on rules because it is important to use common or easily learned tools at each step in the process. Tenor is more procedural, with a templating approach in my implementation.

I have developed a register system, DEXTER (Data EXpression Through Edited Registers),
to help author and manage registers. The register author (RA) uses various editors to define and test the queries the register will use, the data structures that will store the results of those queries, the rules that examine the instances of those data structures, the kinds of things that can be said, the rules determining what things the data allows to be said, the rules that determine the order of things to be said, and the templates that will be used to put data into the document. The RA role may be assumed by an individual or a by group of individuals working on different parts of the register. DEXTER assumes that the original data is in a relational database (miniSQL) and processes the RA's specifications into a Java program [23, 39].

1.4 Related Work

The literature of language generation is voluminous. I cite only what is particularly informative, related or useful.

1.4.1 Terminology

In descriptive linguistics, a register describes an instance of communication as a point of intersection along three axes:
Field: the domain of discourse.

Mode: the form of the discourse, as well as whether the communication is spoken or written, spontaneous or planned, and so on.

Tenor: the effect of the purpose of the communication on the content, form and expression used in the communication.

Control of more registers marks greater sophistication in using a language [24, 28].

1.4.2 Overview of NLG

Hovy and Reiter have each worked extensively in Natural Language Generation (NLG). Both have worked developing NLG systems: Reiter on IDAS and Hovy on RST [31, 70]. Both have given overviews of NLG, Reiter both independently and in conjunction with Robert Dale [32, 67, 68, 69]. As described by these authors, NLG architectures typically possess modules addressing the following tasks:

1. Text Planning: Build an internal semantic representation of data, choose the data and determine the structure of the text.

2. Sentence Planning: Join the text plan with the data to plan individual sentences. Choose key words, determine when to use referring expressions and so on.

3. Surface Realization: Turn sentence plans into grammatical text.

Modules for morphology and formatting are often added [67]. There are three salient features common to NLG systems organized along these lines. First, processing is usually in a “pipeline”, proceeding from module to module in a linear fashion. Second, the text-planning module almost always integrates content determination and rhetorical planning [31]. Choice of data to communicate and organizational planning occur simultaneously. Finally, most systems allow post-generation modification of the text.

Both Reiter and Hovy also distinguish between NLG systems having a planning component and text generation systems lacking a planning component. NLG systems use a range of mechanisms to generate text, from simple templates to much more abstract feature-based systems. Feature-based systems aggregate minimal alternative expressions such as negative
question present, indicating a question should be generated using the present tense of “to be” and with a negative sense.

Limitations in NLG systems include: inability to build a general-purpose sentence generator; limitation of our ability to plan text on more than a toy scale, and the dedication of any NLG system to a single purpose. Underlying problems include lack of theory to guide: lexical selection, sentence planning, discourse structure, domain modeling, and choice between alternative generation strategies. NLG systems suffer from at least two disadvantages versus simple text generation systems: they require an internal data model and they tend to be based on formalisms unfamiliar to most programmers.

Significant immediate or imminent advantages of NLG systems versus simpler systems include: maintainability in the face of changing requirements; higher quality text; multi-lingual output capability; ability to handle multimodal documents that incorporate tables and graphics etc., as well as text more smoothly; ability to incorporate standards (for example the simplified English of AECMA [53]) thus generating conforming documents.

Hooy predicts that NLG systems in the following application areas should soon show success: database content display; expert system recognition; limited report and letter writing; presentation planning in multimedia human–computer interaction; automated summarization and speech generation.

NLG systems and approaches supporting this overview include among others: FUF [16], IDAS [70], JOYCE [66], PENMAN [27, 44] and SPOKESMAN [47], ANA [37], TEXT [46], TAILOR [59], MUMBLE [42], KPML [4], and RST [31]. In Chapter 6 I place computational registers and DEXTER with respect to this general overview of NLG and text generation.

**Hybrid Systems**

NLG methods and templating methods are not exclusive. Hooy, Reiter and Dale acknowledge the practical advantages of hybrid systems that plan text and use some form of templates to realize some or all of the text [32, 68, 69]. Pianta and Tovena examine this kind of hybridization [62]. They treat templating systems as using a single level of representation (strings) and NLG systems as using a multi-level approach, generally text planning, sentence planning and linguistic realization. They argue that linguistic realization should be further divided into sentence grammar, morphological and phonological representations.
In this view a flexible template is a template that can mix the three constituent representations that make up linguistic realization as well as the string representation. Pianta and others describe HTPL (Hyper Template Planning Language), which supports flexible templates [9].

1.4.3 A Few Generation Systems

IVORY is a tool developed by Musen and others to help physicians write progress notes [8]. Two significant, related goals guiding the design of IVORY were: to be acceptable to physicians in the clinical setting and to incorporate physicians into the design process. IVORY uses a graphic user interface to allow physicians to specify a possible clinical finding as present or absent, and if present, as mild, moderate or severe. Findings can also be marked as subject to aggregation when positive, aggregation when negative, or not subject to aggregation. Findings can have both positive and negative names as well as associated SNOMED III codes [14]. Post-editing of output is allowed. IVORY suffers from several limitations. First, the meaning of terminology inserted into the system is not controlled, nor is there evidence of that goal in the discussion of IVORY’s design. Second, IVORY is a dedicated, single-purpose tool relying on many assumptions. IVORY assumes a particular format for progress notes4. Within that format, IVORY assumes that a review of (bodily) systems is desirable. IVORY assumes that positive findings should always appear before negative findings. IVORY can make many of these assumptions because it is a dedicated tool. My effort has been to try to generalize the building of IVORY-like systems.

Schemata originated with McKeown and are the most popular approach to text planning in Reiter’s survey [67]. McKeown refers to the module responsible for determining content and structure for composing an appropriate text as the strategic component [46]. The output of the strategic component is made into English by the tactical component. In addressing strategic issues McKeown focuses on discourse structure (textual organization) and the effect of focus constraints on text planning. She develops schemata, a form of high level paragraph template of rhetorical devices. An individual scheme represents one pattern of text organization by constraining the rhetorical devices available. The scheme that best fits the data and the goal of generation is chosen and filled. Schemata represent a recursive, high level view of compositional strategies. They can be seen as a kind of meta-template.

---

3A medical coding scheme.
4Progress notes are SOAP notes: Subjective, Objective, Assessment, Plan.
at roughly the paragraph level. McKeown's system translates selected schemata and data into English using a functional grammar due to Kay [36]. Schemata have been extended by Paris [59]. To understand and use schemata and the functional grammar requires technical knowledge in both linguistics and computer science. I want to reduce the formal linguistic understanding required while achieving comparable results.

**ANA and Fluency**

Kukich discusses ANA a tool for generating stock reports [37, 38]. ANA has four processing components: fact generator, message generator, discourse organizer and surface generator. Processing is pipelined. Surface generation is accomplished by using inferred messages, a phrasal lexicon and a grammar to build and combine clauses. ANA's processing roughly follows the lines of processing in computational registers. Kukich describes ANA as an example of *knowledge-based report generation*.

Kukich uses ANA as a basis to look at fluency in a natural language generator [38]. She describes fluency and examines how ANA achieves and fails to achieve fluency in its output. In Kukich's terms, fluency skills and fluency defects are categorized as discourse, grammar, syntactic, lexical and semantic skills (or defects.) Discourse skills include: appropriate use of pronouns, ellipsis and hyponyms5, variety in sentence length and consistency in verb tense. Discourse defects include: overly long sentences, overuse of pronouns, missing ellipsis or hyponymy and lack of parallelism. Grammar skills are: effective combination of clauses and appropriate use of conjunction. Grammar defects are: redundant locatives and redundant temporal information. Semantic skills are: choosing interesting messages, appropriately combining messages and filtering details. Semantic defects are: poor message choice, missing a kind of message, and redundancy due either to failure to account for temporal information or lack of causal knowledge. Other skills include appropriate syntactic choice, lexical variety and metaphorical usage. Overuse of syntactic forms and repetition of terms are other defects. According to Kukich, fluency in a knowledge-based report generator is tied to three design principles: knowledge based engineering; macro-level knowledge representation6; integration of situational (domain) knowledge and grammatical knowledge. Kukich found that ANA is fluent in use of: pronouns, ellipsis, hyponymy, varying word choice, producing sentences of varying length and a consistent verb tense. ANA does not

---

5"Ellipsis" is leaving out unnecessary words. A "hyponym" is a less specific reference.

6The system deals with phrases to generate text, rather than building sentences from individual words.
exhibit parallelism. ANA can have trouble placing adverbial clauses, can overuse a single syntactic form or word, can generate a poor or incomplete message, and early on could exhibit anacolutha, inconsistency or incoherence of syntactic form.

Kukich notes two senses of incompleteness in ANA-generated stock reports. First, ANA does not deal with all the events that might be interesting on a given day, nor does ANA place the given day in some trend. This kind of incompleteness was a matter of implementation. Kukich noted a second, deeper incompleteness. While a human author would bring world knowledge external to the stock market into a report about the stock market, ANA can not. To do so would require ANA to have a representation of that world knowledge. Kukich's description of fluency in generated documents is more precise though consistent with the principles of correct natural language generation enumerated in [51]. I address the fluency of registers in Section 6.1.3.

1.4.4 Communicative Goals

Goffman is a linguist who has described system constraints and ritual constraints on communication [22, 30]. System constraints are applicable to all discourse\(^7\). System constraints include four Gricean Norms: truthfulness, relevance, quantity and quality [25]. Ritual constraints are the social rules and practices that determine how a communicator in some community should go about satisfying system constraints. Goffman tells us that good human communicators satisfy system constraints while adhering to community expectations. It follows that a document-generation system should produce output that satisfies system constraints in a way that adheres to community expectations.

It is easier to understand system constraints when thinking about two humans conversing, or even writing back and forth. One system constraint, non-participant signals, the means to include or exclude an individual in a discourse, seems to have no useful interpretation for documents beyond various means of specifying an intended audience. The other systems constraints can be summarized (with generated documents in mind) as follows. The open/close constraint states that how topics are introduced and abandoned is important. Communication includes backchannel signals, non-verbal (non-textual) means of communication. Turnover signals are required to indicate an exchange of communicative roles. For the one-way communication in a document turnover signals mean anticipation of reader

\(^7\)In the sense used here “discourse” refers to virtually any communication, spoken or written.
concerns. Communication must be acoustically understandable and interpretable; documents must be understood physically and semantically. Digressions should be marked by bracket signals.

System constraints also include four Gricean Norms that are standards all communication should meet. They are summarized as follows:

Relevance: Each part of a communication should be relevant. "Relevance", for documents, means avoiding information overload stemming from inclusion of unrelated or tangentially related information.

Quantity: Include only necessary words and information. The right level of verbosity or terseness is also important. Quantity in these senses is important for presentation documents.

Truthfulness: Express only what is believed to be true. A related, more enforceable concept for documents is trust and openness. A document must engender the trust of the reader, or the system which created the document will not be used.

Clarity: Communication (including documents) should be organized, free from ambiguity and obscurity.

1.4.5 Plausibility

Reiter describes a consensus NLG architecture; he also addresses whether that architecture is psycholinguistically plausible [67]. Kukich is also interested in whether systems as psychologically accurate [38]. The issue is whether a system or approach to generation of language reflects how humans generate language, and whether and how such psychologic conformance is important. I address the issue of psychological plausibility as it relates to document generation in Section 2.1, and particularly in Section 6.1.3 after explaining registers thoroughly in chapters 2-5.

1.5 Summary

Hovy, Reiter and Dale are not alone in having reviewed NLG. Mykowiecka among others has also reviewed NLG methods and systems [51, 52]. The summary presented here is consistent with those reviews. One point is notably absent from most discussions. I do not find much said in the NLG community about software packages designed to help build generation systems. While systems are necessarily built to be domain-specific, another
point usually not addressed is discussion of domain-independent generation architectures. Another common feature of NLG systems is that they rely on methods, concepts and terminology that are beyond the knowledge of most programmers. At least some members of the NLG community recognize that highly specialized methods impede acceptance and use of NLG technology. Not enough enterprises have adequate expertise to use the methods [68]. A further consequence of specialized methods is that non-technical domain experts seem to be relegated to the role of a source for knowledge engineering. They do not help build NLG systems. A final point missing in the general discussion is a consistent way to judge the quality of either generation systems or of generated documents. Individual authors have made only initial steps in this direction [38, 51, 68]. Ideas on general communicative goals, particularly Gricean norms, have been interpreted for generation by several authors and the interpretation used as a generation system goal [15, 60].

My research described here makes several contributions. I describe registers independently of any domain and its associated content. I demonstrate that different registers can generate documents in different domains, and that registers can be implemented using standard programming tools. My approach to registers is guided by a desire to see non-technical domain experts directly participate in authoring generation software. I show that registers may be built with software tools that can simplify register authoring and can help manage sets of registers. Finally, I describe registers in terms of fluency, as described by Kukich, and in terms of Goffman constraints, which apply beyond generated documents.
Chapter 2

The Computational Register Architecture

The computational register architecture describes one specific organization for generating documents. The architecture defines generation components, processes and their interactions. Viewing registers at the architectural level allows freedom to imagine varying approaches to implementation. I discuss my approach to implementing the architecture and the implementation itself in Chapters 3 and 4.

2.1 Architectural Goals

Computational registers provide a meaningful solution to the problem stated in Section 1.2. A meaningful solution is one that meets these goals:

1. The approach must support building documents that are suitable for presentation of data.

2. The approach cannot be overly restrictive.

3. Standard methods of computer science must be sufficient to implement the approach.

4. The approach must be as straightforward as possible so that domain experts can help define document generators in their own domain.
5. The approach must lend itself to systematic management of generation software, so that software systems can be created to help design, implement and maintain document generation programs.

The first goal states that an architecture has to describe a means of producing acceptable documents. The second goal means that any constraints on the kind of input data or the meaning of “document” must not restrict applicability of the approach to a narrow set of domains. The set of methods must be as domain-independent as possible. If one these two goals is not met the approach does not work or is not a general approach.

The third goal states that a good approach uses standard methods of computer science. We want as many people as possible to be capable of building a generator (no matter whether the generator is a computational register or not.) Document generators are software tools. Therefore a minimum requirement is that the person building a generator have some technical understanding of programming issues\(^1\). Many people, working in all kinds of enterprises, can write programs. Were we to raise the bar so that a register author is required to have formal knowledge of linguistics, function as a grammarian, or learn new and subtle formalisms, then we would severely restrict the pool of potential register authors. An approach to generation must be straightforward to a technically trained mind, and the methods required should be in standard the canon of computer programming.

There are significant advantages if the domain expert rather than the technical staff of a given enterprise author registers. The domain expert understands the perspective of the audience. Technical staff must struggle to do so by talking to a domain expert in a process of knowledge acquisition. The domain expert is heavily vested in the right presentation of the data. He is more likely to anticipate or notice problems, especially as they appear over time due to evolving needs. He is more likely to be able to determine which problems are important enough to need correction. The domain expert/author is likely to be a user of the register. There are obstacles, however. The domain expert probably has minimal to no programming skill. Acquiring programming skill is unlikely, as the domain expert has other priorities. The fourth goal of the computational register architecture is to involve the domain expert more directly in planning and building a generator. The goal itself is obvious, but how to achieve it is not obvious.

\(^1\)It is possible that a team could build a generator for some domain. In that case someone on the team must have an understanding of programming issues.
The computational register architecture meets the third and fourth design goals in several ways. A register can be constructed from tools and methods within the abilities of many programmers. There are no esoteric methods required that would dramatically decrease the number of programmers who could help build single registers or even register systems. The organization of computational register processing and structures is straightforward. The organization is simplified by carrying out the four top-level processing steps in strict linear order. Linear processing is particularly attractive when we want individuals with no programming background to help in building tools; recursion and backtracking, if we used it, would be difficult for non-programmers.

A register is designed to write deliberately in the sense of Gabriel [21]. Once published, the document may not be altered, nor can the audience interact with the author. From the perspective of the document audience all that matters is whether the text is good and readable. From the perspective of the register author, how the generator works matters a great deal. The register author is creating software that authors. The register author is a “meta-author.” Registers are “psychocompositionally plausible”: they process data in much the same way that people write (see Chapter 6.) I can better support the inclusion of non-technical domain experts in register authoring by extending the use of “standard methods” to composition methodology as well as programming methodology. A meta-author writing a register will be more likely to succeed because the register itself follows a plan similar to how people are taught build documents. The author is already familiar with the process.

Goal 5 above states that an approach should be convenient for systematic management. Registers are a description of a domain-independent approach to document generation, so the issue is larger than building a single register. Sound engineering requires software register systems to help build and maintain multiple registers. Particularly if domain experts, lacking much programming skill, are to be brought into the register-authoring process, then building a register requires more software support than an editor and a compiler. As I note at the end of Section 1.4.2, the idea of using software to build generation tools has not received adequate attention from the NLG community.

Having discussed the goals underlying the design of registers, I can now give a more detailed description of registers.

---

2I am adapting the term “psycholinguistic plausibility” from [67].
2.2 Field

A register begins generating a document by carrying out field processing. The input is the data to be presented, obtained by queries on the DSS. The output is a representation of the data appropriate for building a document. Formally, field processing is a data transformation that maps the external data representation (external scheme) to an internal representation called the internal scheme. The result of field processing is an instance of the internal scheme. The field of a register consists of all the components involved in field processing. Those components are as follows.

Queries. Registers are designed to work with data stored in systems that support queries. If the data is stored in a relational database, then the queries of the field are SQL queries on the database. Obtaining the results of these queries is the first step in examining the data. For the register itself, the external scheme is the scheme of the database. For the field, the external scheme is the set of attributes determined by the register’s queries.

Ontology. The user (who wants to use the data) views the data as representing certain entities. The ontology defines the data structures used by the register to represent domain entities. For instance, if the register is implemented in an object-oriented language such as Java or C++, then the ontology is a set of Java (or C++) classes. The register will use instances of the structures defined in the ontology, rather than the raw data, to build the document. Those structures must be capable of representing both domain entities and presentation information (such as grammatical attributes of domain entities.)

Internal Scheme. The ontology defines a collection of structures that can be used to represent the data expressed in the results of the queries. The collection of possible instances of these structures forms the internal scheme of the register. The internal scheme is distinguished from the external scheme of the data on which we evaluate the original queries. When field processing is complete and the data are represented in instances of the structures defined by the ontology the, collection of resulting data structures is an instance of the internal scheme. For example, if the ontology is a set of Java classes, then the internal scheme is the collection of Java objects that could be instantiated from those classes (a set of variable declarations), and those objects
created at runtime and given value during field processing are an instance of the internal scheme. I will use “internal scheme” for “instance of the internal scheme” when that usage is clear.

**Field Mapping.** The field mapping specifies how query results are embedded in the internal scheme. In my approach to registers the mapping is rule-based (see Chapter 3.)

**Vocabulary.** When a human gathers data and presents it, he uses certain terms to describe the data. It is often necessary to clarify terms used. A production-quality software presentation cannot escape this requirement. A register *vocabulary* records meanings of all terms that might appear in the final output of a given register without having been taken directly from the raw data. For instance, if a register is capable of stating that “John is tall”, and “tall” is not a term from the original data, then the register vocabulary should define the meaning of “tall” when applied to humans in the current register. The vocabulary feature is defined in the architecture so that the user can query the register for the meaning of any unfamiliar or important terms.

Example 2.1 provides an example of raw data represented in an object of the internal scheme. Suppose a database contains the height and weight of an individual as 176 pounds and 69 inches. Assume there is some object called *person* in the internal scheme. The internal form of *person* depends on the ontology. Any of the following would be acceptable transformations, though a single register would use only one.

**Example 2.1** Some acceptable internal data representations

1. `person.height = 69, person.weight = 176`

2. `person.height.ft = 5, person.height.inches = 9, person.weight = 176`

3. `person.height = "average", person.weight = "husky"

4. `person.body_mass_index = 25.8, person.weight = "slightly overweight", person.height = "average"`

□
The field may be defined to transform the data values as required by the document. Literal string values, such as "husky" in example 2.1, should be defined in the register vocabulary. A concise description of field processing is $F : S \rightarrow I$, where $F$ is the field mapping, $S$ is the set of instances of the external scheme, and $I$ is the set of instances of the internal scheme based on the ontology. The mapping instantiates $I$ as a set of structures by filling those structures with values from the data or terms from the vocabulary.

### 2.2.1 Constraints in Field Processing

People do not use software unless they trust it will deliver reliable results. To support a user’s trust a register must do more than define its terms through a vocabulary. *Data-carrying*, particularly combined with a vocabulary, allows a user to verify that a register is reliable. Data-carrying occurs when the raw form of query results, or what the underlying DSS would have shown the user, is made available to the user along with the generated document. In an environment that supports hypertext it is easy to pass the original data along as the document is generated and make it available as a linked document. For instance, when John is described as “tall”, the vocabulary can define the term, and data-carrying allows the user to recover the actual value. Registers are not reversible in general unless they implement data-carrying. For instance, if a register represents John’s height of 75 inches as “tall” there is no way to recover “75 inches” unless the register carries the value along as it generates the document and then makes that value available.

The steps of register processing occur in linear order. Any deviation from linear processing must be confined within a step, such as within the field. There is no constraint at this level on how a field queries the data, except that it must use a method compatible with the DSS. Queries could be formulated and evaluated based on the results of previous queries, or all queries could be formulated initially. However, the field must complete processing before any subsequent processing step. Finally, the field encapsulates access to the DSS and its data representation. Subsequent steps may not access the data directly.

Field processing develops an internal representation of the data found in a set of query results. The re-representation of the data supports building a document. Just as the same queries could be used to generate another kind of document, the instance of the internal scheme can be used to generate another document. Given one register, a second register can be built to re-use some or all of the first register’s field, but process field output differently.
2.3 Mode

Mode processing follows field processing as a register builds a document. The input is the (instance of) the internal scheme as determined by the field, along with the set of constituents defined for the particular register. The output is an outline of the document. The outline is a list consisting primarily of constituents with variables of the internal scheme bound as arguments to the constituent parameters. The register mode consists of all the components involved in examining the internal scheme, selecting messages (constituents), assigning values to selected messages' parameters, and ordering the selected messages. The document outline is a very precise sense of the "form" in descriptive registers. Mode itself is further divided into two distinct but closely related steps: mode-content processing and mode-structure processing.

2.3.1 Mode-Content

Mode-content processing (content processing) accepts the instance of the internal scheme determined by the field, as well as the content messages defined in the register as input. Its output is a set of selected content messages, each with parameters bound to internal scheme values. Order is unimportant in the output. It is natural to treat content determination as part of mode rather than of field. The primary content of a document is the relations it expresses on its representation of the data. These relations constitute what can be asserted about the data. Scripts (due to Schank) and schemata (due to McKeown) are forms that are filled with values to convey meaning [46, 72]. In registers, constituents (messages) describe the potentially interesting relationships on the data and only lack a few details to complete their meaning. For instance, suppose Victor and Elizabeth occur in our data, and further suppose they are married. How we represent the marriage relation has as much to do with what our document is about as the fact that it is Victor and Elizabeth who are married. In one case we may decide that the areMarried message will express Victor and Elizabeth are married. In another case we may have the relation represented as marriage into a family: Elizabeth married into the March family through Victor March. The document content is defined by the message forms.

The mode-content (content) of a register is the collection of components involved in content processing, as follows.
**Content-message constituents.** Everything that may appear in the final document appears only through association with a document constituent. Examples include text, formatting, and graphics. A content message is a constituent describing relations on elements of the internal scheme (as opposed to describing document structure or formatting.) Any constituent is identified as a name with zero or more parameters. Each constituent that ultimately appears in the document outline is rendered in the final document. Rendering expresses the meaning of the constituent. Constituent expression may be a sentence parameterized by data, a sentence fragment, a paragraph worth of text, an image, canned text, a paragraph break, and so on. Thus \textit{areMarried(}Spouse1, Spouse2\textit{)} might be a content message that can receive \textit{Victor} and \textit{Elizabeth} as arguments.

**Content Mapping.** The content mapping is defined on the content messages of the register and the register's internal scheme. The content mapping binds values in the instance of the internal scheme to available content messages. As with the field mapping, I think of the content mapping as a sequence of rules, although it is not necessary to implement the mapping with rules. The mapping considers internal scheme values and available messages in rule conditions. The consequent of a rule binds internal scheme values to content-message parameters, which is called asserting the message. Content processing consists of applying the content mapping.

Formally, \( Mc : I \mapsto C \), where \( Mc \) is mode-content, \( I \) is the internal scheme, represented at runtime by an instance, and \( C \) is a set of content messages with parameters bound to structures in \( I \). \( C \) is a set because no content message is repeated unless with different values for its parameters. The order in which the messages appear, even the number of times they appear, is as yet undetermined.

At the end of Section 2.2 I state that the field of one register can be re-used in building another register. One way to process the field results differently is to use a different set of content messages. For example, consider electronic medical records. A register might examine those records to build a patient summary for a physician. The same queries and internal representation might be used with different content messages to summarize information for the patient or the insurance company. The same data treated differently at any point in processing results in generating a different kind of document.
2.3.2 Structure

Mode-structure processing (structure processing) begins when content processing ends. Input to structure processing consists of the set of content messages asserted (output) by content processing, the presentation messages defined for the register, and a subset of the internal scheme determined by the field. The output of structure processing is an ordered list of constituents (either content or presentation messages) mixed with instructions. Instructions may make certain constituents unavailable, may alter the foci of the register, and may close contexts within the register. (Register foci and context are described below.) The ordered list of constituents and instructions constitutes the document outline. Content messages represent semantic information in the document, presentation messages represent information about document structure, and instructions affect processing during document generation. For example, the content message \texttt{areMarried(Victor, Elizabeth)}, is semantic, while the \texttt{leadParagraph()} message describes part of the document form. An instruction such as close \texttt{leadParagraph} means that “lead paragraph” no longer describes the document context, which must be represented during processing. Register \textit{structure} is comprised of the components involved in structure processing:

\textbf{presentation-message constituents.} A constituent that is not a content message is a presentation message. Definition of presentation messages and content messages differs in only two respects. Presentation messages are defined to represent structural and formatting information in the document, such as beginnings, paragraphs, titles, and so on. Further, every content message defines a context. Presentation messages may or may not define a context. Presentation messages have zero or more parameters. Structure processing determines the actual parameters of presentation messages. Argument values are either internal scheme values or literal values such as canned text.

\textbf{context.} In computational registers a context is a string name that represents a logical location in the document. A context name corresponds to a specific constituent name, either a content message or a presentation message that describes document structure. A set of contexts may belong to a named context class. Some constituents may be blocked altogether from certain contexts or classes of context. Whether a given context or an instance of a given class of context is open can affect the way a
constituent is eventually expressed in a document. Multiple contexts may be open simultaneously.

**focus.** A focus is an attribute of a document that may have different values at different points in a document. (Context could be considered a special case of focus, since the current context is a document attribute whose value may change.) For instance, the subject under current discussion might be “Victor March”, or the subject under discussion might be “Raphael Winkler.” If “Victor March”, then in, “He wrote the book”, the subject pronoun “he” is understood to refer to “Victor March.” In building a document a focus is used to represent information that must cross message boundaries.

**instruction.** Instructions in document outlines represent that a specific context should be closed, that a particular focus should be set (or unset), or that a specific message that might otherwise be available for use in subsequent processing should be blocked from further use.

**epistemic scheme.** The portion of the internal scheme made up of only Boolean values is called the epistemic scheme. The most common use of a Boolean in the internal scheme is to record that some relationship is known to exist. For instance: hasSpouse is true, isRich is false. The structure mapping, discussed below, may examine only Boolean values in the internal scheme. Content processing encapsulates all other direct examination of the internal scheme. The structure mapping may bind arbitrary internal scheme values to presentation messages.

**structure mapping.** The structure mapping is defined on all constituents of the register and the epistemic scheme. It determines a sequence of content messages (already given arguments), presentation messages and their arguments, and instructions. Again I think of the mapping as rule-based. The conditions of the rules consider the available constituents and the epistemic scheme. The consequent of a rule may bind values to parameters of presentation messages (hence asserting the message), reassert content messages, and issue instructions. Structure processing consists of applying the structure mapping.

Example 2.2 shows a segment of a document outline. In that example, title is a (context) presentation message that may be parameterized with members of the internal scheme
object person. LeadParagraph is a context presentation message with no parameters. InitialDescription is a content message that can take person as an argument. Subject is a focus that may be given value by InitialDescription.

Example 2.2 A segment of a document outline

1. Assert title(person.FirstName, person.LastName)
2. Close context title
3. Assert LeadParagraph
4. Set Subject
5. Assert InitialDescription(person)

Line 1 in Example 2.2 asserts title, parameterized with members of person. Line 2 closes the title context explicitly. Line 3 asserts LeadParagraph. The message is rendered as a new paragraph. The context LeadParagraph, an instance of the class of paragraph contexts, is now open. Line 4 indicates that Subject should be set by the next constituent capable of doing so. InitialDescription is asserted on line 5. Since content messages are contexts, the context InitialDescription becomes the second open context. If InitialDescription can set Subject, for example give Subject the value person, it will do so.

Formally, mode-structure is described as $M_s : C \rightarrow O$, where $M_s$ is mode-structure processing, $C$ is the mode-content output, and $O$ an ordered list of constituents, all with necessary parameters appropriately instantiated, and instructions.

Suppose we have built a register describing contents of an electronic medical record for use by a physician in a clinic setting. Structure processing offers another chance to fork off a new register. Perhaps the same data representation and the same content messages can be rearranged into a form suitable for hospitalized patients. The same data and the same relations on data may be expressed in a different outline resulting in a different document.

The concept of mode is the composition of the mode-content and mode-structure mappings: $mode : I \rightarrow O$. 

2.4 Tenor

The final top-level step of register processing is tenor processing. The input is a document outline and a set of possible expressions for each constituent in the register. The output is a document in some document representation language such as HTML. The tenor controls issues such as word choice and sentence structure. The tenor of a register consists of components related to the processing.

Expressions. Expressions are the mechanism used to render a constituent in the document. Each constituent defined in the register has an associated list of possible expressions. If the constituent has parameters, the expressions may use the actual parameter values. Expressions may also make use of context information or focus information. Use of particular expressions may be restricted to specific contexts or certain kinds of contexts, the value of a focus may be expressed directly or indirectly, or the focus may only be used to determine the gender of a pronoun. A given constituent may have only one expression or several. Details of how expressions are implemented are not part of the register architecture. The architecture separates the concept of expression of a constituent from the concept of constituent because different registers may find it convenient to use different expressions for the same message, and different register systems may find it convenient to use different methods to express constituents.

In symbols we can describe expression as a mapping associated with a constituent: 
\[ E_c : \langle p_c, f, ctx \rangle \rightarrow L, \] 
where \( E_c \) is an expression of constituent \( c \), \( p_c \) a possibly empty list of actual parameters to the constituent \( c \), \( f \) a set of focus values, \( ctx \) the current context information, and \( L \) a sentence of the representation language.

Representation Language. A register generates a document represented in some language such as HTML, ASCII text, or RTF [13]. It is acceptable for a single register to target more than one representation language. If more than one language is allowed, then the tenor (and each expression) must know which to use.

In a register that begins with a physical description of a person, the outline segment shown in example 2.2 might be expressed in a number of ways, including those shown in example 2.3, which assumes HTML as the document representation language.
Example 2.3  Some possible expressions of the outline from Example 2.2

<H1>About Victor March</H1>
<P>
Victor March is an old, slightly overweight male.

Applying an alternative (equivalent) expression for the InitialDescription message might result in:

<H1>About Victor March</H1>
<P>
Victor March is an old man. He is slightly overweight.

In a different register that uses the same messages and shares this portion of the outline but with a different set of expressions, we might find the expression shown in example 2.4.

Example 2.4  Alternative expression

<H1>Victor March, Cat Owner</H1>
<P>
Victor March owns one cat, Carmen.

In any of the cases of Examples 2.3 and 2.4, since the Subject focus is set to Victor March the next message expression may elect to use “He” in place of “Victor March”, since Victor March is known to be the subject. The register would violate convention, however, if the first reference to Victor March in the text of the lead paragraph were “he.”

The idea of building a new register from an existing register by providing an alternative at each processing step works for tenor processing as well. Let’s continue the patient-record example. It may be that the same data and the same thoughts organized in the same way can be expressed, but differently, to the physician and to the patient, or to the generalist and the specialist. If the first register addresses the physician, it might use standard medical terminology, many abbreviations and show relatively complete test
results. A second register could take the same information and use common language, avoid abbreviations, and show fewer or only descriptive results of tests to a patient.

The more processing shared between registers the more similar their resulting documents. Two registers that share only the field may result in quite different documents, while two registers that share all but the expressions of the register's tenor will be very similar.

Finally, $T : O \mapsto D$, where $T$ is tenor processing, $O$ is the document outline received from mode-structure processing, and $D$ the final output expressed in the representation language.

### 2.4.1 Embedded Descriptions of Documents

Searching through a number of electronic documents by hand takes too long to be feasible in most cases. Using software can give crude results that both miss pertinent occurrences of the target and include false hits. The prime example is trying to find the right document on the World Wide Web. A similar problem is extracting an answer to specific question from one of a possibly large set of documents as in [1].

Generated documents should give the appearance of natural language. However, they do not contain natural language; they contain carefully and automatically constructed language. Computational registers begin with raw data, represent it appropriately for presentation, and embed that data in a document outline by associating it with significant named relations. As the register builds the document, a progressively more detailed description of the final document is formed. Registers should be able to represent various levels of description (meta-information) clearly and embed that information in the final document representation. For example, if a web site uses documents generated by registers and publishes the descriptive scheme, then the generated documents are searchable at a higher level than allowed by examining the text alone. The descriptions can be represented as *semantic tags*, and the documents can be semantically searchable by any search engine able to understand the tags. In principle, these tags could implement the various forms of metadata described by Bohm: metadata for representation of media types, content description or classification, document composition, history and location [7]. Applications range from making document search results more precise to ensuring that a concept is not accidentally “published” by having pieces of the concept appear in several different documents. I have only made minimal experiments to show, for example, that the names of data fields
and constituents and descriptions of ensuing text can be carried in an HTML document as an easy result of generation. I have not investigated how to represent that information to make it most useful to automatic search engines. I mention semantic tagging, however, because registers support consistently embedding arbitrarily precise descriptive information into documents. Generated documents therefore offer one solution to the problems associated with a multitude of electronic documents and relatively imprecise automated search strategies.

2.5 Summary

Registers process data to generate a document. If \( R \) denotes a register, then \( R : d \rightarrow D \), where \( d \) is data contained in the scheme of a DSS, for example the scheme of a database, and \( D \) the generated document. It is worth noticing that the first step of the mapping is a restriction of the domain of field processing to the scheme of a set of query results, the external scheme of the field. Breaking down the process reveals:

\[
R(d) = T(Ms(Md(F(Query-Results(d))))) = D
\]

where \( R \) is the register, and all other letters have the meaning noted above. The process is linear. Figure 2.1 is a schematic representation of the process.

Data \[\rightarrow\] Field \[\rightarrow\] Mode-Content \[\rightarrow\] Mode-Structure \[\rightarrow\] Tenor \[\rightarrow\] Document

Figure 2.1: Schematic of register processing.

Register processing is not reversible in the absence of data-carrying. The register may alter the representation of data in a way that results in ambiguity going backwards. For example, a height of 69 inches in the data be expressed as “average height” in the internal scheme. The internal scheme may represent any height value of 66 inches through 71 inches in the same way, making it impossible to precisely determine the original data value.

I can now put more detail into Figure 1.3:
Field

Mode

Mode Structure

Tenor

Raw Data as in a database

Instance of Internal Scheme

Set of Content Message Constituents

Document Outline Constituents and Instructions

Final Document Representation

Legend:

Processing Step

Component supporting processing

Input and Output

Indicates flow of data

Indicates processing support

Figure 2.2: Register details, components and processing.
Chapter 3

Design Decisions

The computational register architecture discussed in Chapter 2 describes register components and processing. It does not specify how components or processing steps should be implemented. Anyone wishing to build a single register or a register system must begin by specifying design decisions: how each component will be implemented; how each process will be implemented; and any constraints required beyond those mentioned in Chapter 2. In this chapter I describe the decisions underlying my own first register system, DEXTER. I first bring out two important design decisions: use of rules and use of templates. I then give an overview of design for each segment of the architecture. Chapter 4 discusses the implementation.

3.1 Registers are Rule-based

Text planning in registers is carried by field and mode processing. In my approach, each of these steps relies on a mapping defined by rules. The most general form of the rules is: if conditions then action. The conditions are always defined on the inputs to the processing step and the action helps to build the output of the processing step. The precise meaning of conditions and actions varies with each processing step. Details appear in subsequent sections of this chapter.

Rules are an attractive method for processing because they are in the standard canon of tools for computer scientists, and their if-then structure and semantics are easily understood even by those not trained in programming. Rules are also well suited to this set of tasks because each of the text planning sub-tasks (building the internal scheme, selecting and
parameterizing constituents, and putting those constituents into order) can be carried out in a piecewise fashion, with one or a few rules per "piece" of the eventual construction.

3.2 Procedural Templates to Express Constituents

Field processing transforms the result of data queries to an instance of the register’s internal scheme. Mode-content processing binds values from the internal scheme to constituent parameters, and mode structure places the resulting constituents in a document outline. Tenor processing causes each constituent appearing in the outline to be expressed in turn. Constituents expression is defined by templates.

A constituent represents a unit of meaning in a document\(^1\). Constituents express meaning by association with expressions. A constituent may have multiple possible expressions but has only one meaning. For constituent \(C\) and alternative expressions \(c_i\) and \(c_j\), \(c_i\) and \(c_j\) must be *semantically equivalent* in the document context. For example, consider the five expressions: (1) *Victor and Elizabeth are married*, (2) *Victor is married to Elizabeth*, (3) *Elizabeth is married to Victor*, (4) *Victor married Elizabeth July 4, 1969*, and (5) *Victor and Elizabeth have been married since July 4, 1969*. Expressions 2 and 3 are semantically equivalent. It is hard to imagine the circumstances where 1 is not equivalent to 2 or 3. Expression 5 might be equivalent to 1-3 in some contexts, but in general the inclusion of the date of marriage makes it a different statement. Expression 4 is semantically distinct from all the others because it does not logically imply that Victor and Elizabeth are married at present. Expression 5 could be used in a manner equivalent to the other expressions only if the information that Victor and Elizabeth remain married was conveyed by another expression in close physical proximity to 5. Constituent expressions are implemented as templates. A template is canned text with slots for variable text values and some control information\(^2\). Registers build presentations of data that are familiar to readers. A non-technical register author requires a mechanism to allow him to communicate what data means through the medium of a register. Templates are a good choice because they are

---

\(^1\)The kind of meaning varies depending on the medium of the data the constituent will express. For example, the meaning of a constituent may be *place a graphic specified by a parameter value in the document at a given point*. In text a constituent will be expressed as a clause, sentence or group of sentences using parameter values.

\(^2\)Generated documents are textual. Inclusion of other media is made possible by the ability of the representation language to include instructions for display of other media in the text of the document representation. The entity in in HTML is an example of textual inclusion of graphic media.
macro-level language structures that carry a complete meaning clearly. They look very much like what the domain expert wants to write.

There are three chief concerns with using templates. (1) The term usually connotes something inflexible and likely to produce significant and numerous grammatical errors in text. (2) Lack of textual variety follows from inflexibility. (3) Templates are viewed as hard to maintain. All of these concerns arise from the use of canned text. None of these concerns prohibits using templates successfully.

One approach to reducing these concerns is to take traditional canned text templates and build grammars to generate a fluent expression of each. Processing a constituent amounts to finding and using the right grammar. The idea is analogous to the macro-level clause combining grammars used in ANA [37, 38]. This approach results in grammatical expression, some variability, and easier modification than with the original template. There are two concerns, however. First, the process of building such a grammar requires something like this: The domain expert writes a simple template; a programmer with sufficient skill builds a compatible grammar; the expert verifies the outputs are acceptable. Second, the grammars need to use certain information about both the grammatical status of terms or clauses and the document content, for instance the current focus. The grammars require a good deal of meta information, much of it linguistic. It is better to avoid this extra layer of authoring complexity. We therefore reject this grammar-based approach.

A simpler approach is to treat templates as procedures that process input to compute output. The input is the actual parameters to be expressed, as well as current document state information. (Each is needed to use a grammar-based approach as well.) A template is a collection of expressions of the intent of a constituent, all equivalent in the given presentation. Alternative expressions are selected based on input and simple decision processes specified by the author of the template. For instance, if parameter name represents a person, use the person's name, unless there is information that this person could be referred to by a pronoun. The procedural approach requires as much meta information as the grammar approach but removes the extra technical layer in the authoring process, as any educated person should be able to produce a reasonably grammatical text\(^3\). Allowing selection from multiple equivalent expressions makes a template capable of varying text. The only issue left is maintainability. It is generally held that templates are hard to maintain.

\(^3\)There is no accepted metric for "reasonably grammatical." The best we can do is something like Kukichi's analysis of fluency in [38].
[68]. However, by building a system to manage registers, in the process helping authors find, examine, modify and experiment with templates, we can reduce the burden. Because the approach makes the initial authoring process simpler and more accessible to domain experts, it is reasonable to try the procedural approach instead of the grammar-based approach.

3.3 Context and Focus

Two kinds of control information become important as a document is processed and generation decisions must be made. The first kind of information is context, the second focus. Context affects what we want to say and how we say it: A title context is different from the context of a paragraph of text; lead paragraphs are treated differently from subsequent paragraphs which are treated differently from closing paragraphs.

I approach context simply. As required by the architecture, content messages and some presentation messages name contexts. Classes of contexts are, for instance, titles, paragraphs, headings and so on. A context may be recognized as belonging to one or more classes of context if the context name contains the name of the class as a substring. For example, context “LeadParagraph” is a member of the “Paragraph” class, as is “named-Paragraph.” Contexts do not carry data; they are simply string values the register may check as it processes. As the register processes constituents that represent contexts, the context names are put in a list of open contexts. Context information is used in tenor processing. When the tenor processing engine wants to render a constituent it does so by choosing an appropriate expression associated with the constituent. Expressions may be defined to require that a specific context or class of contexts be open. Multiple contexts may be open at once. I treat context as a stack, so a document may move from a given context to another and return. A context may be closed explicitly. When context A closes, all contexts placed above A on the stack also close.

Suppose an expression needs to include a string representing a name in the generated text. Sometimes a referring expression or pronoun is needed and sometimes it is not needed. A focus is a named value that helps with this kind of decision. I allow a focus to be a string literal or a reference to data in the internal scheme. If an expression is about to generate the name of a male person and notices that the value of the focus named subject is this
same person, then the expression can recognize that it is safe to use a masculine pronoun rather than the actual name.

3.4 General Approach, Database and Ontology

A register system is a suite of tools to help register authors create and maintain registers. Registers in turn process data to generate documents. My system, DEXTER, provides editing tools to allow the register author to define the necessary field, mode-content and mode-structure rules, expressions, and supporting structures. Each editor in the system can translate the author's definitions into code in a high-level language, compile and execute that code. Each provides the author control over order of execution of the rules. The system includes a stand-alone engine to process the rules of a register outside the editing environment. I elaborate on DEXTER in Chapter 4.

The data that feeds a register is structured, coming from some source that supports queries. Although it is not required by the architecture, I assume that a register is fed from a relational database. Relational databases are prime examples of the kind of data a register should handle. They are well understood and widely used.

The ontology of a register defines the data representation used by all the processing modules of a register. The structures defined in the ontology must represent both domain information and linguistic information such as grammatical attributes of domain entities. The representation capability must be robust across a wide variety of domains. I treat the ontology as a set of Java classes conforming to some specific conventions [23].

3.5 Field

3.5.1 Rules and Queries

Field rules are defined to examine query results and translate the values found there into an instance of the internal scheme. Translating the results of a set of database queries from one representation to another is the most technically challenging part of building a register.

In my implementation each field rule is associated with one query. A query is defined inside the register system. It is a name, an associated, parameterized SQL query, and a definition of column names and types for the result. A query result may be examined by more than
one rule. I make the simplifying requirement that a field rule may neither explicitly invoke another field rule as part of its action, nor may it parameterize a query. (Queries are parameterized only by arguments passed to the register.) Limiting field rules in this way reduces the complexity of the rules, hence the complexity of authoring the rules. Further, allowing such actions would have increased the initial programming burden of building a register system. Here are three sample field rules:

**Example 3.1 Three field rules**  
Rule 1  
IF (the query named `getPerson` returns exactly 1 row) THEN  

- `FoundTarget` is assigned TRUE  
- `Target.FName` is assigned `getPerson.FName`  
- `Target.LName` is assigned `getPerson.LName`  

Rule 2  
IF (the query `getPerson` returns exactly 1 row  
AND column `getPerson.Sex` is not null )  
THEN  

Assign `Target.Sex` the value `getPerson.Sex`  

Rule 3  
IF ( `Target.Sex` is not assigned ) THEN  

Assign `Target.Sex` the value `UNKNOWN`  

Rule 1 in Example 3.1 is defined on query `getPerson`. Assume `getPerson` seeks a person uniquely identified by first and last name. Then the requirement is that the query return only one row. If `getPerson` does return just one row, then the person is found, and the rule records the name in the internal scheme variable `Target`. The rule also records that it did find the target. `FoundTarget` is an example of an epistemic variable. Subsequent rules may depend on epistemic variables as a shorthand for rechecking a set of complex conditions.  
The second rule in Example 3.1 is typical. This rule requires that the query `getPerson` succeed as expected, as well as that the result column labeled “Sex” be non-null. If these
requirements are met, then the rule assigns Target's Sex field.
The final rule in Example 3.1 is also a common form. It checks only on what is already known in the internal scheme. In this case if the Target.Sex field has not been assigned it is marked as empty. Rule order is significant.
Generally a field rule looks like this:

IF ((nullcheck) AND condition)
THEN action-list

Null values are valid in either the external or internal scheme. The nullcheck is a conjunction of requirements on columns of the query result examined by the rule. The nullcheck is parenthesized above to indicate it should be evaluated prior to the condition, and the condition evaluated only if the nullcheck succeeds. Each conjunct of the nullcheck requires a given column have a null value or have a non-null value. Null values are important because rule actions are not conditional, and actions may need to handle null and non-null values differently at runtime. Columns referenced in an action may be included in the nullcheck automatically. The condition is a general Boolean expression formulated on the data already stored in the internal scheme and on the data in the query result columns. The nullcheck is evaluated before the condition, and if the nullcheck fails the rule fails without evaluating the condition. Lazy evaluation prevents runtime errors, since conditions can reference columns that may be null. The action list is an ordered list of actions to take when the rule’s nullcheck and condition are satisfied. The actions valuate the instance of the internal scheme of the register using the data found in the query result, thereby accomplishing the the data transformation for which the register field is responsible.
More formally, a field rule has the following form:

\[
\text{IF } \left( t.a_1 = \text{Null} \wedge \ldots \wedge t.a_k = \text{Null} \wedge t.a_{k+1} \neq \text{Null} \wedge \ldots \wedge t.a_p \neq \text{Null} \wedge R(\hat{t}, s) \right) \\
\text{THEN } s_q = E_q(\hat{t} \cup s), \ldots, s_r = E_r(\hat{t} \cup s)
\]

Attribute \( a \) of the \( i^{th} \) tuple of a query result is denoted \( t.a_i \). The \( j^{th} \) value of the internal scheme is denoted \( s_j \). Line (1) performs a nullcheck on the results of the query on the external scheme. If line (1) is satisfied then line (2) is evaluated. In line (2), \( R \) is a general condition on the non-null portions of the query result (denoted \( \hat{t} \)), and the current state of the internal scheme (denoted \( s \)). If both lines (1) and (2) are satisfied then line (3),
the action list, is executed. The action list may assign a value to each variable $s_k$ of the internal scheme. The value assigned to $s_k$ is determined by expression $E_k$ defined on $s$ and $f$. String or numeric constants or functions may appear in the expressions of the action list or in the general condition. The fact that rules may depend on values assigned to a variable $s_k$ by some other rule makes the order of rule evaluation important. Requirements such as a result having a single tuple (as shown in Example 3.1 are not explicit in the rule condition. Field rules have a type, and requirements like having a single tuple in the query result are enforced by by the semantics assigned to each field rule type.

3.5.2 Taxonomy of Field Rules

Each field rule examines the result table of exactly one query. There are a number of ways to deal with a tabular result. I have specialized field rules to several distinct types to capture the semantics necessary for each useful approach to tabular results.

**take-first rule.** This type of rule is used when want to process only one tuple in the result, though we know there may be more tuples. A take-first rule evaluates its condition on each tuple in the result until it finds a tuple on which the condition succeeds, or until all tuples are tried without success. As soon as the rule condition succeeds on one tuple, the rule has succeeded; No further tuples are examined. The rule fails only if it fails on all tuples, or if there are no tuples in the result. An example requiring a take-first rule is inclusion of one image in a document. Suppose we have a query that finds images of a person. If we want just one image in the document and do not have sufficient information to pick otherwise, we can use a take-first rule to get just the value found in the first tuple of the result.

**singleton rule.** When only one tuple is expected, a singleton rule can be used. It succeeds only if the result has exactly one tuple and that tuple satisfies the rule condition. An example is finding the current spouse of a married, law-abiding U.S. citizen.

**iterative rule.** An iterative rule processes each tuple in the result table in turn. In programming terms the rule is evaluated in a loop and succeeds or fails on each row. The action of the rule is carried out once for each row on which the condition succeeds. The register author may specify that certain actions are to be taken before the rule executes, only on its first success on each success or only when all tuples have been
processed. An example of an occasion to use an iterative rule is accumulating a list of scores from a set of tuples with the form \textit{StudentName, AssignmentIdentification, ScoreOnAssignment}. The \textbf{Mid Term Student Grade} register discussed in Section 5.1 requires iterative rules.

**aggregate rule.** Another way to process the tuples of a result is to collect information about all of them and treat those results as a single value. For instance, we might not want each value in a particular column, rather the mean of all values in that column. Aggregate rules handle this case. For example, suppose rather than wanting to accumulate a student’s scores, we wish to record the mean of his scores, but only have a query to collect the scores.

**introspective rule.** Like all field rules, introspective rules are formally defined with respect to a specific query. However, an introspective rule does not directly access any part of the query result. The rule is predicated solely on internal scheme values and does not refer to query results directly in any of its actions. Introspective rules tend to rely on epistemic values in the internal scheme, or to distinguish ranges of values already stored in the internal scheme. Using introspective rules with epistemic values is convenient because it lets us avoid repeating complex conditions on query results, and it facilitates forward chaining in the field rules [40]. The third rule of Example 3.1 is an example.

### 3.5.3 Vocabulary

I have made only one design decision about vocabulary beyond what is described in Chapter 1; I have not implemented a vocabulary. Building a vocabulary tool for a register system would be a significant undertaking. It seems possible to use existing tools, for example wordnet for the purpose, so long as the tool contains sense information and is extensible [18, 50]. The primary role of the vocabulary is tangential to the generation process in registers. Vocabulary entries can provide syntactic information about a word, but my design perspective has been that this information could be represented in the ontology\footnote{See Section 6.1.2 however.}. The real role of the vocabulary is to ensure that the eventual document reader can trust the document and its generator. Providing a vocabulary allows the user to query the register
for the intended meaning of terms used. Outside of a production-quality system, or unless fielding a system for user-acceptability testing, this role is not so important. I have deferred implementation of vocabulary management in the first version of DEXTER.

3.6 Mode

Mode processing selects, parameterizes and orders document constituents. I allow constituents to be defined as repeatable or not repeatable. A non-repeatable constituent is unavailable after it has been placed in a document. Repeatable constituents may occur more than once in a document. For instance, a mainTitle constituent should appear only once, while a showAddress constituent might appear several times. Constituents are otherwise handled as described in Section 2.3.

3.6.1 Content Rules

The conditions of content rules are evaluated on the register’s internal scheme and the availability of content messages. A content rule action usually associates some values in the internal scheme with the parameters of a content message. A repeatable content message might also be explicitly blocked, meaning it is marked as no longer available. The result of processing content rules is a set\(^5\) of content message constituents available for inclusion and rendering in the final document. This set serves as input to mode-structure processing. An example of a content rule follows:

Example 3.2 A content rule CRule 1

IF ( Know( Target,\{ FName, LName, Age, Sex, Weight, Height\} ) ) THEN

    assert DescribePerson( Target )

Generally a mode-content rule looks like this:

    IF ( Know( valueset\(_a\) ) AND Don’t Know( valueset\(_b\) )
        AND Know Value( valueset\(_c\) ) AND Know Empty( valueset\(_d\) )
        AND Condition
    THEN
        Action List

\(^5\)No constituent may be repeated with the same parameter values.
Condition is a general Boolean expression over the values of the internal scheme. Value sets \(a\) through \(d\) are possibly empty sets of values from the internal scheme of the register. The Know predicate for a value set is true if and only if for each element of the value set the element is assigned or a value or explicitly marked as empty of value. Predicate Don’t Know is the complement of Know. The Know Value predicate is more specific than Know, requiring each element to be assigned a value rather than marked empty. The predicate Know Empty requires the elements to be marked empty rather than assigned any other value. In content rules the actions of the Action List bind values to constituent parameters or block constituent availability. Constituent argument values may be from the internal scheme or they may be literal values\(^6\). Once a constituent is parameterized with actual values, the constituent is placed in the output list of mode-content processing, and so becomes available for expression in the final document. Binding values to constituent parameters is called asserting the constituent.

Mode content rules may only assert constituents that are content messages. Assertion of messages that cannot be repeated requires some care. Initially all content messages defined in the register are included in a list of available constituents. When a constituent that may only be used once is asserted, it is automatically blocked. A rule does not fail just because one of the constituents it would assert is blocked. Instead the rule succeeds and any actions asserting unavailable constituents have no effect\(^7\). The output of mode-content processing is a proper set; each time a repeatable constituent appears in the output set it must be asserted with different parameters.

Formally mode-content rules are defined as follows:

\[
\begin{align*}
\text{IF} & \quad (\text{know}(s_a, \ldots, s_i) \land \\
& \quad \neg \text{know}(s_j, \ldots, s_m) \land \\
& \quad \text{know}_{\text{value}}(s_n, \ldots, s_p) \land \\
& \quad \text{know}_{\text{empty}}(s_q, \ldots, s_1) \land \\
& \quad \mathcal{R}(s)) \\
\text{THEN} & \quad \text{if available}(C_1) \text{ then assert } C_1(\sigma_1) \\
& \quad \vdots
\end{align*}
\]

\(^6\)If vocabulary were implemented then a literal value passed to a constituent would have to appear in the vocabulary.

\(^7\)The ability to have a rule succeed while some of its actions cannot and do not occur requires some care. In the next chapter I will discuss DEXTER and its mode-content rule editor MORCOND. MORCONED allows the register author to test lists of content rules and see when constituents become unavailable, as well as which constituents were actually asserted.
if available( $C_r$ ) then assert $C_r(\sigma_r)$

where $s$ and $s_i$ are as in the formal statement of field rules in Section 3.5.1 above, $\mathcal{R}$ is again a general Boolean condition, the $C_j$ are constituents parameterized by subsets of the internal scheme represented as $\sigma_j$, available is a predicate that determines if a given constituent name may be used, and assert is a procedure to include a parameterized constituent in the set of meanings possible to express in the document. Any s.b appearing in any $\sigma$ should appear in one of the four epistemic predicates know, $\neg$know, know\textsubscript{value} and know\textsubscript{empty} already described.

The order of rule evaluation should not matter here, but it does: Two rules may be satisfiable that would each assert the same non-repeatable constituent, but with different actual parameters values. The assertion of the rule evaluated first will be available for inclusion in the final document; the alternative assertion will be blocked. This state of affairs suggests that the semantics for asserting non-repeatable constituents are suspicious. In fact, the conditions of the two rules should be exclusive. A register system cannot readily enforce exclusive conditions, however, because the conditions of the rules include general Boolean expressions involving variables and possibly functions. Checking that two such conditions are exclusive before variables and functions are evaluated at runtime is not computationally feasible. An alternative way to eliminate the problem is to make availability of all constituents a rule might assert a premise of the rule. In that case, if an author wishes to assert $k$ constituents in one rule, but is willing to allow any subset of them to be asserted under the same conditions, then he needs to write $2^k - 1$ rules with the same condition, one for each subset. In fact the author is better off to write one rule for each constituent in this case, resulting in only $k$ rules. Rules that assert multiple constituents are intuitively convenient. I have decided that the convenience of being able to assert multiple constituents in one rule is worth the risk of rule order affecting the output of the register. In principle an authoring tool in a register system can help a register author identify a subset of rules that could assert the same non-repeatable constituent. The author can then notice if the constituent would be asserted differently, and if, in that case, the conditions of the rules that would make the different assertions are exclusive in the domain.
3.6.2 Structure Rules

Structure rules are evaluated on the availability of constituents, on the current context as the document is built, and on a subset of the internal scheme. The result of processing a list of structure rules is a list of constituents and processing instructions that forms a document outline. When evaluation of structure rules begins, the available constituents are those content messages asserted by the content rules and the presentation messages defined in the register. Each structure rule may mark any number of constituents as unavailable, so the order of processing in these rules is important. As structure rules are processed, they assert different constituents, and those constituents are placed in a list representing the order of constituent appearance in the final document. Each assertion of a constituent will change the current context if the asserted constituent is defined to represent a context. Structure rules are not allowed arbitrary conditions on the internal scheme. They are allowed to examine only Boolean values (epistemic variables) from the internal scheme. The idea is that a structure rule might need to determine if some combination of values is significant. The field should represent whether the particular values are significant or not by use of an epistemic variable.

A structure rule may take two kinds of action: It can place a constituent in the document outline (assert the constituent) or it can issue a processing instruction. A processing instruction may be used to explicitly block a particular constituent from any further assertion, indicate that the next constituent that can set a given focus should set that focus, or it may explicitly close an open context. A single structure rule may take as many actions as necessary. Example 3.3 shows some structure rules.

**Example 3.3** Two structure rules
SRule 1
IF (Available(aboutTitle) AND FoundTarget) THEN
   assert context aboutTitle(Target FName, Target LName)
   close context aboutTitle

SRule 2
IF (Available(DescribePerson)) THEN
   assert context untitledParagraph()
set Subject
assert content DescribePerson() □

In Example 3.3, Target represents a person. The first rule checks that the presentation message aboutTitle (a context) is available for use and that the internal scheme knows Target was found. If the rule succeeds, aboutTitle is asserted with the first and last names of Target as arguments. The aboutTitle context, opened by the assertion, is immediately closed. The second rule checks only that the content message DescribePerson is available for use (was asserted by the content rules and has not been blocked.) If so, the rule asserts the context untitledParagraph (that will fail silently if that message is unavailable), issues an instruction that the next message which can should set the focus Subject, and asserts the content message DescribePerson. If DescribePerson can set the focus named Subject it will do so.

Constituents are defined to be repeatable or not. However, in structure processing I look only at the names of the content messages produced by content processing. I do not distinguish distinguish between the same message with two different sets of parameters. Making this unique-names assumption is restrictive, but it simplifies writing rule conditions. Otherwise it must be possible to ask if a given message is available and if its parameters satisfy further conditions.

The general from of a structure rule is relatively complex:

\[
\text{IF ( Available( Constituent}_i \text{ ) and ... and Available( Constituent}_j \text{ )} \\
\text{and not Available( Constituent}_k \text{ ) and ... and not Available( Constituent}_l \text{ )} \\
\text{AND Open( context ) AND ( ISBoolean}_p \text{ and ... and ISBoolean}_q \text{ )} \\
\text{THEN} \\
\text{Action List}}
\]

The Constituent\(_n\) are either content messages or presentation messages, context is a named context or class of context, and ISBoolean\(_m\) is an epistemic value in the internal scheme. One point to note is that a rule may check one particular context, or it may check for an instance of a context class (see Section 3.3.) Action List is a sequence of assertions or instructions, as described above.

\(^8\)In principle another message, for example a subtitle, could be asserted in the AboutTitle context.
Document Trees

Structure rules are evaluated on a tree. The nodes of the tree are the structure rules. The evaluation engine tries to reach a leaf node in depth-first fashion. Each node that succeeds appends its actions to the document outline. Reaching a leaf node (traversing a branch) completes processing and completes the document outline. Backtracking occurs when a rule fails. (Backtracking is allowed within processing steps.) Three special kinds of nodes are allowed: a cut node, to prevent backtracking, a success node, indicating a branch is traversed, and a failure node, indicating an unrecoverable error has occurred. A document tree represents a family of possible forms for a particular kind of document. Each form corresponds to one branch. Examples of document trees are provided and discussed in Section 4.2.5, for example in Figures 4.25 and 4.26.

I allow backtracking out of necessity. Evaluating structure rules in a list rather than a tree requires conditions can recognize the subsequent failure along a branch and thereby cause the current rule to fail. Such conditions must subsume the conditions of succeeding rules, and become unmanageably large.

3.7 Templates and Expressions in the Tenor

I explain my motivation for using templates to express the meaning of constituents (hence to generate text) in Section 3.2. A template is associated with a constituent and two lists: a list of named parameters and a list of expressions. When a constituent $C$ is found in the document outline template, $T_C$ is also found, and the actual parameters of $C$ passed to $T_C$. Templates must also receive context and focus information. A template examines its expressions, chooses one, and uses that expression to generate text in the final document.

A given expression may require a particular context or kind of context to be open, may require given parameters to be non-null (or null), and may require a particular focus to be set. If these requirements are not satisfied then the expression is not used. If a template parameter is allowed to be null then there should be an expression that does not require that parameter be non-null. Expressions may incorporate processing by a set of defined procedures, for instance distinguishing if a parameter value is singular or plural, and adjusting output text accordingly. The template in the following example has two expressions.
Example 3.4 Template for DescribePerson(Person Target)

Expression 1:

require context: any paragraph context
require non-null: Target.FName, Target.Name, Target.Age, Target.Sex
require null: none
Foci: if focus Subject unset set focus Subject to Target
Text: Target.FName Target.LName is a Target.Age year old

Expression 2:

require context: any paragraph context
require non-null: Target.FName, Target.Name, Target.Age
require null: Target.Sex
Foci: if focus Subject unset set focus Subject to Target
Text: Target.Fname Target.LName is Target.Age years old.

If one of the contexts (or context classes) required by an expression is open, all required foci have a value, and all parameters are null (or non-null) as required, then the expression is called an applicable expression. Any method of selection may be implemented when more than one expression is applicable. DEXTER chooses an arbitrary applicable expression.

3.8 A Register System: Management via Editors

I assume that the primary users of software for document generation are large enterprises with a data-storage system used by a number of different populations, each with its own document conventions. In that case there are two problems. One problem is how to build a single register. The second problem is how to manage a number of registers. The two problems can be approached simultaneously by building a register system: a set of software tools to build and maintain registers.

3.8.1 Authoring one register

A register system can help register authoring by providing a suite of tools to edit registers. Each distinct component or process in the register has its own editing module. If a register
is authored by a team, then the more technical members can use focused tools to tackle
the more technical register aspects, such as defining the field. Less technical members
can use more general tools to build more general (less technical) portions of the register,
such as document structure and constituent expressions. Each editor should define one
kind of component (query, vocabulary entry, ontology class content message constituent,
presentation message constituent or expression) or the rules for a single processing step
(field, mode content, mode structure.) In my approach, Java classes represent the ontology,
so it makes sense for the rules and other components to be implemented in Java.

Rule editors must offer views of the components they rely on. The field-rule editor must
show the author the shape of query results and the classes of the internal scheme. It must
allow the author to create a new object in the internal scheme (a new variable) and write
rules associated with a given query. The register author does not use the editors to write
Java; he uses them to give a high-level specification of the rule he wants. The specification
is translated into Java. The author must then be able to execute the rule (Java program)
and examine its results without leaving the editor. Consider the example of field rules.
To build a set of field rules it is important the author be able to specify a list of rules
to execute and be able to test that list. Since field rules are based on queries, the field
rule editor should allow the author to execute the register queries and see the results. The
principle is that the the field rule editor should allow the author to specify all the necessary
parts of a rule, make it executable, and test it either alone or in conjunction with other
rules. The same principle applies to mode content and mode structure rule editors.

An editor for components must allow the author to see underlying and related components
and to name and specify the attributes of the component. For example, the expression
editor must allow the author to choose a constituent, recognizing whether that constituent
is a content or a presentation message, recognizing whether it is a context or not, recog-
nizing the the parameters of the constituent, and recognizing whether the constituent has
previously defined expressions. The editor must also exhibit the register’s contexts, the
context types, the foci and stored procedures defined for the system. The author must
than have the facilities to define the expression in terms of these basic components, canned
text and stored procedures.
3.8.2 Managing Several Registers

An editor that can support authoring as described above can be extended to perform the following actions in support of authoring a set of registers:

1. Definition of one rule, component or register based on an existing definition of another rule, component or register.

2. Searching for all rules or components in a register that satisfy a given condition and performing some operation on all items found.

3. Alerting the author to potential problems, such as no field rule being defined that could assign a value to a particular internal scheme variable, no possibility of asserting a defined constituent, and so on.

The first action means a register system can support inheritance between registers. Supporting inheritance can reduce duplication of authoring tasks from one register to the next and ensure consistency between related registers.

The second kind of action has many forms, such as, finding all the field rules that assign a given internal scheme object, finding all the content rules that assert a given content message, or finding all the structure rules which operate on a given focus. If an object is deleted from the internal scheme, then it is important to find the rules that assign to that object, remove the assignments, and regenerate the modified rules. Updating expressions can be assisted by finding all expressions in all constituents that contain a particular string.

The third kind of action can be helpful in any processing or definition step. For instance, a register editing system of the kind I have described can examine a list of field rules and notice which rules do not appear in the list, and insure than none appear twice. It can examine the rule list and see if there is an internal scheme object that is never assigned in any of these field rules. An expression editor can check that some expression associated with a constituent uses each parameter of the constituent. Similarly, the editor can automatically determine which subsets of possible parameter combinations are not covered by any associated expression when null parameters are allowed in a constituent.

A system that can perform these kinds of tasks eases building a single register or managing a set of registers. Technical and nontechnical authors can work on a register in concert because authoring tools are modularized. Any approach to document generation should allow for tools that perform these kinds of basic tasks.
3.9 Issues and Restrictions

I have described registers in Chapter 2 and made my approach to registers more concrete in this chapter. In the next chapter I discuss my specific register system, but I first address some issues and restrictions.

Since Java classes represent domain entities, it makes sense that register rules and other components be implemented in Java. As a result, the various editing tools need to translate the author's input into Java, which must be compiled and interpreted. Executing a register is a matter of executing a Java program.

DEXTER's plan contains simplifications. There is no vocabulary mechanism. I examined using wordnet [18, 50] as support for a vocabulary component. While that appeared feasible, it would have increased the scope of programming required. The purpose of my work is to define an architecture, show that it works, and that it can be managed by software tools, all according to the goals set forth in Section 2.1. Implementing vocabulary control is not necessary for that purpose. Vocabulary control addresses reliability and user trust in a given domain; it is primarily a consideration for a production tool, or for testing user acceptance once the approach is shown to produce reasonable documents.

Data-carrying, like vocabulary management, can be deferred. Keeping the query results as a serialized Java object while post-field processing is carried out is technically simple. Assuming HTML as the target document representation language makes it similarly straightforward to provide a hyperlink or links in the final document that accesses those tables as preformatted text. However, data-carrying should be implemented in a production-quality system, or in a system developed to test how well users trust generated documents.

Text generation requires an internal representation of data to represent syntactic information. The classes of the ontology all implement a single interface that allows objects of the internal scheme to be queried for their grammatic attributes. I say more about the DexterObjectInterface in Chapters 4 and 6.

Finally, although registers should embed a representation of the constructed document, I did not believe I could manage the process intelligently before I built and evaluated a functional register system. I therefore decided not to implement semantic tagging as proposed in Section 2.4.1 in the first iteration of DEXTER.

Programming a register system is a large undertaking. For practical reasons I decided to implement the critical elements of the system: field rule, mode-content rule and structure-rule
editors, an expression editor, tools defining content and presentation-message constituents, a tool to allow definition and setup of a register, and an engine to process a completed register. I deferred the tasks of building a query editor and a class (ontology) editor. It is simple enough for research purposes to provide the necessary queries and classes by hand, adhering to guidelines defining what I conservatively assume an editor for parameterized queries or classes could do.
Chapter 4

DEXTER: An Implementation

DEXTER stands for Data EXpression Through Edited Registers. DEXTER implements the architecture described in Chapter 2 using the approach described in Chapter 3. DEXTER is not a complete production level register system. It currently lacks an editor, for instance. However, the critical pieces for showing what registers can do are all in place. After providing a brief overview of DEXTER and some notes on platform, language and so on, I provide an extensive example of using DEXTER to build a register. The example allows me to show how DEXTER works and to discuss most of DEXTER’s features.

4.1 Overview and Platform Issues

4.1.1 Platform

DEXTER is implemented, using Sun’s Java Workshop, as a Java application. It runs under Sun’s Solaris operating system. DEXTER’s database support is provided by Mini SQL version 1.0.11 from Hughes Technology and the Imaginary mSQL.0.9.6 JDBC driver written by George Reese [39].

DEXTER is a large project. At present it consists of roughly 35,272 executable Java statements (judged by counting semi-colons) spread over about 115 class files. These figures cover the editing tools as well as the built-in classes DEXTER uses to support writing registers. There are an additional 10 configuration files with a total of 204 lines of configuration information. All of these figures are somewhat inflated because I have not re-used code in many places that I could have. I chose to avoid code re-use at times
because I was not certain what the requirements of the next step would be. DEXTER could be re-built in about 20,000 lines of Java code.

4.1.2 Overview

DEXTER is a set of tools, mostly editors, for building and maintaining registers. DEXTER implements editors for Field rules (FRED), Mode rules (MORCONED for content rules, STRUCTED for structure rules), constituents (CONED for content messages, PRESED for presentation messages) and expressions (CODEX.) DEXTER also provides a stand-alone register-execution engine called EXECUTOR. The level of functionality provided by these tools varies somewhat, as I discuss below. A schematic of DEXTER as implemented is provided in Figure 4.1. Once a register is created (via ADMIN) the register author may begin by using FRED, CONED, or even PRESED. Only after STRUCTED has created a document tree may EXECUTOR be used to execute a register outside the editing environment.

---

Figure 4.1: Schematic of DEXTER as implemented.
As Figure 4.1 indicates, both FRED and CONED must be used before MORCONED. MORCONED and PRESED must also be used before STRUCTED. CODEX, however, may be used when either one of CONED or PRESED have been used. Figure 4.1 shows dependencies among DEXTER’s tools. Those dependencies do not necessarily reflect the processing pipeline of a DEXTER register. A register is required to process in a linear fashion, but the register author may design sections of the pipeline in a non-linear fashion, so long as he does not violate the necessary dependencies.

DEXTER does not implement any vocabulary management, hence there is no vocabulary editor. My reasons for deferring implementation of vocabulary management are discussed in Chapter 3. DEXTER also lacks a query editor and lacks an ontology (class) editor. The current version of DEXTER exists primarily to show that registers as described in Chapters 2 and 3 can be built for a variety of domains using the same software package to author the registers. Because the query and class editors would offer relatively little information about the feasibility of building registers or a register system, I have chosen not to implement them to this point. I did implement a small tool, ADMIN, to perform some of the function of these two tools.

Given the number of tools comprising DEXTER and the many tasks to be carried out in building a register, the best way to look into DEXTER’s details is to build a register using DEXTER.

### 4.2 Building a Register

A register is a software tool that expresses certain information about given data in the form of a document. The document should account for both user expectation and data values. I demonstrate building a register called Reg_Tutor with DEXTER in the rest of this chapter\(^1\).

#### 4.2.1 Pre-processing: data, corpus, queries, classes

To create a register, the register author begins by understanding the data, understanding what the register document should look like, and understanding how the data supports the document. To do this he must examine the scheme of the data as it is stored in the

---

\(^1\)This register is essentially the Pet Scan register Victor Marek developed using DEXTER.
database and look at the known data to see what values are likely. He creates a corpus of sample documents to serve as a guide and a measuring stick as he creates the register. Finally, he examines these documents to see which data values underlie the document, and to determine the queries that he can use to get that data. The results of these queries is a set of tables. The register evaluates data in these tables, which form the external scheme of the register field, and create its own internal representation of the data during field processing. Field processing is implemented by a set of field rules in DEXTER. The author uses FRED to write those rules.

Data

The register author does not normally choose data; it is given to him. Here the data at hand is contained in five tables:

**TABLE PEOPLE:**

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
<th>Age</th>
<th>Sex</th>
<th>HtIn</th>
<th>WtLb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arnold</td>
<td>Random</td>
<td>36</td>
<td>M</td>
<td>67</td>
<td>156</td>
</tr>
<tr>
<td>Brenda</td>
<td>Chan</td>
<td>24</td>
<td>F</td>
<td>62</td>
<td>102</td>
</tr>
<tr>
<td>Natalia</td>
<td>Sturek</td>
<td>20</td>
<td>F</td>
<td>64</td>
<td>105</td>
</tr>
<tr>
<td>David</td>
<td>Kramer</td>
<td>28</td>
<td>M</td>
<td>69</td>
<td>182</td>
</tr>
<tr>
<td>Elaine</td>
<td>Webster</td>
<td>44</td>
<td>F</td>
<td>65</td>
<td>174</td>
</tr>
<tr>
<td>Frank</td>
<td>Patrick</td>
<td>52</td>
<td>M</td>
<td>73</td>
<td>186</td>
</tr>
<tr>
<td>Gina</td>
<td>Patrick</td>
<td>51</td>
<td>F</td>
<td>63</td>
<td>136</td>
</tr>
<tr>
<td>Hal</td>
<td>Rashman</td>
<td>22</td>
<td>M</td>
<td>69</td>
<td>167</td>
</tr>
<tr>
<td>Victor</td>
<td>March</td>
<td>56</td>
<td>M</td>
<td>69</td>
<td>176</td>
</tr>
<tr>
<td>Elizabeth</td>
<td>March</td>
<td>51</td>
<td>F</td>
<td>64</td>
<td>120</td>
</tr>
<tr>
<td>Joseph</td>
<td>Oldman</td>
<td>40</td>
<td>M</td>
<td>67</td>
<td>180</td>
</tr>
<tr>
<td>Dianna</td>
<td>Hobart</td>
<td>35</td>
<td>F</td>
<td>63</td>
<td>110</td>
</tr>
<tr>
<td>Miroslaw</td>
<td>Szczyielski</td>
<td>42</td>
<td>M</td>
<td>72</td>
<td>180</td>
</tr>
<tr>
<td>Helena</td>
<td>Szczyielska</td>
<td>41</td>
<td>F</td>
<td>68</td>
<td>145</td>
</tr>
<tr>
<td>Beth</td>
<td>Silverstein</td>
<td>44</td>
<td>F</td>
<td>66</td>
<td>120</td>
</tr>
<tr>
<td>Raphael</td>
<td>Winkler</td>
<td>45</td>
<td>M</td>
<td>69</td>
<td>140</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WtLb</th>
<th>Eyes</th>
<th>Hair</th>
<th>MaritStat</th>
</tr>
</thead>
<tbody>
<tr>
<td>156</td>
<td>brown</td>
<td>brown</td>
<td>S</td>
</tr>
<tr>
<td>102</td>
<td>brown</td>
<td>black</td>
<td>M</td>
</tr>
<tr>
<td>105</td>
<td>brown</td>
<td>black</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>blue</td>
<td>red</td>
<td>M</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
<td>-----</td>
<td>----</td>
</tr>
<tr>
<td>182</td>
<td>blue</td>
<td>red</td>
<td>M</td>
</tr>
<tr>
<td>174</td>
<td>green</td>
<td>blonde</td>
<td>D</td>
</tr>
</tbody>
</table>

... | brown | black | M  |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>186</td>
<td>brown</td>
<td>black</td>
<td>M</td>
</tr>
<tr>
<td>136</td>
<td>green</td>
<td>brown</td>
<td>M</td>
</tr>
<tr>
<td>167</td>
<td>blue</td>
<td>brown</td>
<td>M</td>
</tr>
<tr>
<td>176</td>
<td>blue</td>
<td>gray</td>
<td>M</td>
</tr>
<tr>
<td>120</td>
<td>blue</td>
<td>blonde</td>
<td>M</td>
</tr>
<tr>
<td>180</td>
<td>green</td>
<td>blonde</td>
<td>M</td>
</tr>
<tr>
<td>110</td>
<td>blue</td>
<td>blonde</td>
<td>M</td>
</tr>
<tr>
<td>180</td>
<td>brown</td>
<td>brown</td>
<td>M</td>
</tr>
<tr>
<td>145</td>
<td>brown</td>
<td>brown</td>
<td>M</td>
</tr>
<tr>
<td>120</td>
<td>brown</td>
<td>brown</td>
<td>M</td>
</tr>
<tr>
<td>140</td>
<td>brown</td>
<td>black</td>
<td>M</td>
</tr>
</tbody>
</table>

**TABLE PETS:**

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
<th>Kind</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arnold</td>
<td>Random</td>
<td>dog</td>
<td>Daisy</td>
</tr>
<tr>
<td>Brenda</td>
<td>Chan</td>
<td>Collie</td>
<td>Frenchie</td>
</tr>
<tr>
<td>Brenda</td>
<td>Chan</td>
<td>Boxer</td>
<td>Frank</td>
</tr>
<tr>
<td>David</td>
<td>Kramer</td>
<td>cat</td>
<td>Cuddles</td>
</tr>
<tr>
<td>Elaine</td>
<td>Webster</td>
<td>cat</td>
<td>Kalamazoo</td>
</tr>
<tr>
<td>Elaine</td>
<td>Webster</td>
<td>dog</td>
<td>Big boy</td>
</tr>
<tr>
<td>Hal</td>
<td>Rashman</td>
<td>canary</td>
<td>Sylvester</td>
</tr>
<tr>
<td>Clark</td>
<td>Lawson</td>
<td>Collie</td>
<td>Frenchie</td>
</tr>
<tr>
<td>Clark</td>
<td>Lawson</td>
<td>Boxer</td>
<td>Frank</td>
</tr>
<tr>
<td>Debbie</td>
<td>Kramer</td>
<td>cat</td>
<td>Cuddles</td>
</tr>
<tr>
<td>Victor</td>
<td>March</td>
<td>cat</td>
<td>Carmen</td>
</tr>
<tr>
<td>Elizabeth</td>
<td>March</td>
<td>cat</td>
<td>Carmen</td>
</tr>
<tr>
<td>Beth</td>
<td>Silverstein</td>
<td>badger</td>
<td>Stalwart</td>
</tr>
<tr>
<td>Raphael</td>
<td>Winkler</td>
<td>badger</td>
<td>Stalwart</td>
</tr>
</tbody>
</table>

**TABLE MARRIED:**

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
<th>SFName</th>
<th>SLName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brenda</td>
<td>Chan</td>
<td>Clark</td>
<td>Lawson</td>
</tr>
<tr>
<td>Clark</td>
<td>Lawson</td>
<td>Brenda</td>
<td>Chan</td>
</tr>
<tr>
<td>David</td>
<td>Kramer</td>
<td>Debbie</td>
<td>Kramer</td>
</tr>
<tr>
<td>Debbie</td>
<td>Kramer</td>
<td>David</td>
<td>Kramer</td>
</tr>
<tr>
<td>Gina</td>
<td>Patrick</td>
<td>Frank</td>
<td>Patrick</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>Frank</td>
<td>Patrick</td>
<td>Gina</td>
<td>Patrick</td>
</tr>
<tr>
<td>Hal</td>
<td>Rashman</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>Hal</td>
<td>Rashman</td>
</tr>
<tr>
<td>Victor</td>
<td>March</td>
<td>Elizabeth</td>
<td>March</td>
</tr>
<tr>
<td>Elizabeth</td>
<td>March</td>
<td>Victor</td>
<td>March</td>
</tr>
<tr>
<td>Dianna</td>
<td>Hobart</td>
<td>Joseph</td>
<td>Oldman</td>
</tr>
<tr>
<td>Joseph</td>
<td>Oldman</td>
<td>Dianna</td>
<td>Hobart</td>
</tr>
<tr>
<td>Miroslaw</td>
<td>Szczysielski</td>
<td>Helena</td>
<td>Szczysielska</td>
</tr>
<tr>
<td>Helena</td>
<td>Szczysielski</td>
<td>Miroslaw</td>
<td>Szczysielska</td>
</tr>
<tr>
<td>Raphael</td>
<td>Winkler</td>
<td>Beth</td>
<td>Silverstein</td>
</tr>
<tr>
<td>Beth</td>
<td>Silverstein</td>
<td>Raphael</td>
<td>Winkler</td>
</tr>
</tbody>
</table>

**TABLE LIKES_ANIMALS:**

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
<th>Kind</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Arnold</td>
<td>Random</td>
<td>dog</td>
<td></td>
</tr>
<tr>
<td>Arnold</td>
<td>Random</td>
<td>cat</td>
<td></td>
</tr>
<tr>
<td>Brenda</td>
<td>Chan</td>
<td>dog</td>
<td></td>
</tr>
<tr>
<td>Brenda</td>
<td>Chan</td>
<td>cat</td>
<td></td>
</tr>
<tr>
<td>David</td>
<td>Kramer</td>
<td>cat</td>
<td></td>
</tr>
<tr>
<td>Elaine</td>
<td>Webster</td>
<td>cat</td>
<td></td>
</tr>
<tr>
<td>Hal</td>
<td>Rashman</td>
<td>fish</td>
<td></td>
</tr>
<tr>
<td>Hal</td>
<td>Rashman</td>
<td>cat</td>
<td></td>
</tr>
<tr>
<td>Frank</td>
<td>Patrick</td>
<td>dog</td>
<td></td>
</tr>
<tr>
<td>Victor</td>
<td>March</td>
<td>cats</td>
<td></td>
</tr>
<tr>
<td>Elizabeth</td>
<td>March</td>
<td>dogs</td>
<td></td>
</tr>
<tr>
<td>Joseph</td>
<td>Oldman</td>
<td>fish</td>
<td></td>
</tr>
<tr>
<td>Beth</td>
<td>Silverstein</td>
<td>badgers</td>
<td></td>
</tr>
<tr>
<td>Raphael</td>
<td>Winkler</td>
<td>badgers</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE PHOTOS:**

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>Oldman</td>
<td>Photos/joldman.jpeg</td>
</tr>
<tr>
<td>Dianna</td>
<td>Hobart</td>
<td>Photos/msoldman.jpeg</td>
</tr>
<tr>
<td>Victor</td>
<td>March</td>
<td>Photos/vmarch.jpeg</td>
</tr>
</tbody>
</table>
The register author must understand the data. Here the data describes a set of people, their current spouse (if any), their pets (if any) and their animal preferences (if known.) Fortunately, the attributes names are not mysterious. The table PEOPLE includes common physical attributes, as well as marital status (W = widowed, S = single, D = divorced and M = married.) It is possible to be both widowed and married, but our data only tells us the most recent status acquired. The combination of first name – last name is assumed to be a unique identifier. The table PHOTOS gives a file location for a digital image of some of the people. The register author must know how to find the photo files from the given paths.

The table MARRIED relates tuples in the PEOPLE table. The married table is symmetric; if Brenda Chan has a spouse Clark Lawson, then Clark Lawson has a spouse Brenda Chan. Nobody has two spouses.

The table PETS describes the pets owned by each person, both by name and by a string identifying the kind of animal. If two people are married and one owns a pet then the data insists that the spouse also owns the same pet. Finally, the table LIKES ANIMALS describes the kinds of animal each person likes. In both of these tables two tuples may agree on the FName LName attributes because one person may own more than one pet, and one person may like more than one kind of animal.

For most of these tables we make a closed-world assumption [71]: If a name is not in the PEOPLE table it is nowhere else; if a person is married we know it; if a person owns a pet we know it. The exception is in LIKES ANIMALS, where the information is known to be incomplete. In this case, we assume we know all the likes if we know any, but if someone does not have any entry in the LIKES ANIMALS table we do not know what it means. A final point the register author would need to consider is that there may be null values in the data. A few attributes (the first and last names) are not allowed to be null. In PETS the register author may assume that both Kind and Name will have non-null values.

The part of a register that deals with the data as stored in the database is the Field. To be capable of building a register’s field the author must understand the database. He should be able to read these definitions:
# 'PEOPLE' Table structure
#
CREATE TABLE PEOPLE (     
    FName CHAR(16) NOT NULL,  
    LName CHAR(16) NOT NULL,  
    Age INT,  
    Sex CHAR(1),  
    HtIn INT,  
    WtLb INT,  
    Eyes CHAR(12),  
    Hair CHAR(12),  
    MaritStat CHAR(1)  
)

# 'PETS' Table structure      # 'MARRIED' Table structure
#
CREATE TABLE PETS (  
    FName CHAR(16) NOT NULL,  
    LName CHAR(16) NOT NULL,  
    Kind CHAR(12),  
    Name CHAR(16)  
)

CREATE TABLE MARRIED (    
    FName CHAR(16) NOT NULL,  
    LName CHAR(16) NOT NULL,  
    SFName CHAR(16),  
    SLName CHAR(16)  
)

# 'Likes_Animals' Table structure
#
CREATE TABLE LIKES_ANIMALS (    
    FName CHAR(16) NOT NULL,  
    LName CHAR(16) NOT NULL,  
    Kind CHAR(12)  
)

# 'PHOTOS' Table structure
#
CREATE TABLE PHOTOS (       
    FName CHAR(16) NOT NULL,  
    LName CHAR(16) NOT NULL,  
    LOCATION CHAR(64)  
)

The Corpus

Given a good understanding of the data, the register author must decide what the output document should look like, what goals it should achieve, and how it should achieve those goals. In this case the initial goal is a very brief (one paragraph) document that describes an individual, mentions the pets he (or she) and spouse (if any) own, and then describes the
animal preferences of the individual, as well as those of his or her spouse if the individual is married. The register author creates the following group of sample documents (or "corpus") by hand:

(P1) Victor March is an old, slightly overweight male of average height. He and his wife, Elizabeth, own a cat named Carmen. Victor likes cats, but Elizabeth prefers dogs.
PHOTO OF Victor March CENTERED HERE

(P2) Joseph Oldman is male, middle-aged and slightly overweight. He and his wife, Dianna Hobart, do not own any pets. Joseph likes fish. Dianna's preferences are not known.

(P3) Miroslaw Szczygielski is a middle-aged, slightly overweight male who is tall. He and his wife Helena Szczygielska do not own any pets. Their pet preferences are unknown.

(P4) Beth Silverstein is a middle-aged woman of average height. She and her spouse Raphael Winkler own a badger. Both Beth and Raphael like badgers.

(P5) Natalia Sturek is a single young woman, slim and of average height. She does not own any pets, but likes lions.
PHOTO OF Natalia Sturek CENTERED HERE

Several points merit comment. First, these paragraphs serve a dual role. They help the register author know what the output ought to look like as he develops the register. Second, he may judge the quality of the output of the final register by its (intuitive) proximity to these paragraphs. In paragraph (P1) there is complete information, and Victor March's photo is included because it is available. (Spouse photos are not included.) In paragraph (P2) information is complete except there is no photo available. Dianna Hobart is the first instance where no pet preferences are known. In (P3) lack of pet ownership is explicitly noted. Paragraph (P4) is one instance of aggregation in a register: Both Beth and Raphael like badgers (probably fortunate.) Only in (P5) does the register author finally deal with someone who is not married. Natalia Sturek's photo is available, hence included. The register author intends that a spouse surname is omitted when it agrees with the surname of the individual of interest (see paragraph (P1)) but included otherwise (as in (P2), (P3), (P4).) Finally, the paragraphs show reasonable variation in the expression of equivalent facts.

With this corpus available the register author has a guide to work from and a measure to evaluate his results. He also has an idea of what sentences, or paragraphs in larger
document, he needs, and what data goes into those sentences. To create the document he needs to formulate the queries to gather the data that goes into the sentences, define the ontology and objects of the internal scheme to represent the data, and define the rules that use the query results to instantiate the internal scheme. He further needs to define the constituent messages to represent the sentences, write rules to bind the variables of the internal scheme to those messages, and define expressions for those messages in terms of the message parameters. Finally, he will need to define rules to order those messages which have received arguments, so that applicable expressions of the messages, representing the data as indicated by the sentences in the corpus, may be placed in the output document.

Queries

The first use for the corpus is to allow the register author to determine exactly the data he needs and how to get that data. Let’s agree to refer to the individual of interest as the person of focus (POF). The register author assumes that the register is parameterized by the first and last names of the POF. He writes queries in terms of POF.Fname, POF.Lname. (DEXTER allows parameters to be passed to a register at runtime.)

Examining the corpus, the register author notices that he will need to verify that the requested individual (POF) exists in the database. Further he will need descriptors of the age, height and weight of this person. He will also need to know if the person is married. He derives the following query on the PEOPLE table:

(Q1: getPerson )
SELECT FName, LName, Age, Sex, HtIn, WtLb, MaritStat
FROM PEOPLE
WHERE FName = POF.Fname AND LName = POF.Lname

Executing this query with “Victor March” as the POF yields these results:

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
<th>Age</th>
<th>Sex</th>
<th>HtIn</th>
<th>WtLb</th>
<th>MaritStat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Victor</td>
<td>March</td>
<td>56</td>
<td>M</td>
<td>69</td>
<td>176</td>
<td>M</td>
</tr>
</tbody>
</table>

Under the assumptions noted about the data, this query will always have 0 or 1 rows in its result. The name getPerson will identify the query in the register.

Now the register author must find the spouse of the POF:
(Q2: getSpouse)
SELECT SFName, SLName
FROM MARRIED
WHERE FName = POF.Fname AND LName = POF.Lname

Here the result, assuming Victor March again, is again one row:

+-----------------------------+
| SFName        | SLName      |
+-----------------------------+
| Elizabeth     | March       |
+-----------------------------+

The register author must also gather information on pets:

(Q3: getPets)
SELECT Kind, Name
FROM PETS
WHERE FName = POF.Fname AND LName = POF.Lname

The result may be zero to an unknown number of rows, though the March family has only one pet:

+-----------------------------+
| Kind          | Name       |
+-----------------------------+
| cat            | Carmen     |
+-----------------------------+

For instance, the couple of Brenda Chan and Clark Lawson have two pets. A query that may have more than one tuple result must be treated differently in the field mapping than one with a single tuple result. The register requires information about the animal preferences of the POF, and it requires an image of the POF if one is available.

(Q4: getPetKindsLiked)
SELECT Kind
FROM LIKES_ANIMALS
WHERE FName = POF.Fname AND LName = POF.Lname

(Q5: getPhoto)
SELECT LOCATION
FROM PHOTOS
WHERE FName = POF.Fname AND LName = POF.Lname

These queries have the following results in this particular case:
<table>
<thead>
<tr>
<th>Kind</th>
<th>cats</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>Photos/vmarch.jpeg</th>
</tr>
</thead>
</table>

Finally, the register author needs the analog of query (Q4) for the spouse of the POF. Here he encounters a limitation of registers as I have defined and implemented them, and a limitation of the MiniSQL version on which DEXTER is based. First, only the POF name is passed to the register. For our example the name is Victor March. Registers do not allow queries to be parameterized except by values passed in from the command lines. Registers cannot formulate new queries once invoked. Therefore, the register author cannot use the value found for the spouse name by query (Q1) to find the pet preferences of the POF’s spouse. MiniSQL (in version 1.0.11) does not allow nested queries. The solution turns out to be query (Q6):

(Q6: getSpousePetsLiked)
SELECT LIKES_ANIMALS.Kind
FROM LIKES_ANIMALS, MARRIED
WHERE MARRIED.FName = POF.FName and MARRIED.LName = POF.LName
AND MARRIED.SFName = LIKES_ANIMALS.FName
AND MARRIED.SLName = LIKES_ANIMALS.LName

The result for Victor March on query (Q6) is:

<table>
<thead>
<tr>
<th>Kind</th>
<th>dogs</th>
</tr>
</thead>
</table>

The register author has not needed either the PEOPLE.Eyes or PEOPLE.Hair attributes. At this point, using a full-featured register system, the author would have developed these queries in a query editor and in the process would have created the Reg_Tutor register. Since DEXTER does not include a query editor, query development is done by hand. A query editor needs the following information from the author:
1. Number and type of arguments to the register, in this case two strings. These arguments parameterize the queries.

2. The name of the database to query (or names if the query editor and register system support registers from multiple databases simultaneously — DEXTER does not.)

3. For each query a name, the text of the query, referring to the command line arguments, and the type of the resulting columns unless the query editor can determine this from the database.

Since DEXTER does not implement a query editor, ADMIN provides the needed functionality to create a register. In DEXTER it makes sense to defer creating the register until the classes that the register will use are identified.

The Classes of the Reg_Tutor Ontology

The register author specifies members of two classes in this case:

```java
tutIKindAnimal {
   string IKind
   string IAnimalName
}

tutIPof {
   string ILName
   string IFName
   string IAge
   string IWeight
   string IHeight
   string ISex
   string ISLName
   string ISFName
   array string IAnimalsPofLikes
   array string IAnimalsSPofLikes
   array tutIKindAnimal IAnimalsOwned
}
```

The register author prefixes each class and member name with "I" to help distinguish that it is part of the internal representation. I show these class names prefixed by tut because they are part of the DEXTER tutorial register, Reg_Tutor. A dialog with the author is sufficient to generate a useful class. The initial information required is the name of the
class and the name and type of each member. This step is currently carried out by hand because DEXTER does not implement a class editor. DEXTER requires that each class implement the DexterObjectInterface and be built by aggregating members of types already defined in the specific register's ontology or in the DexterOntology. For each class member \( X \), DEXTER requires a \( getX \) function and a \( setX \) function.

**The DexterOntology.** DEXTER provides a number of fundamental classes. The classes all implement the interface DexterObjectInterface. Some wrap a Java class: DexterBoolean, DexterInt, DexterFloat, DexterString. Others wrap a pair of Java classes: DexterStringInt, DexterStringBoolean, DexterStringFloat. DexterDate is a simple date class. DexterArray wraps an array of DexterObjectInterface objects.

**DexterObjectInterface.** The DexterObjectInterface is a Java interface: a specification of methods that must be implemented by each class implementing the interface, and a number of defined constants. An object of classes supporting presentation must represent data and presentation information. Certain grammatical information must be known to reference an object value in a text. The DexterObjectInterface restricts grammatical information to number (one of SINGULAR, PLURAL or UNCERTAIN), gender (one of MASCULINE, FEMININE, NEUTER or UNCERTAIN) and count\(^2\) (one of COUNT, NONCOUNT or UNCERTAIN.) The DexterObjectInterface also supports internal names for objects (not strictly necessary given that Java implements the reflect API), and explicitly marking objects as empty and or assigned. The interface also supports two methods to represent an object as a string: identity(), for showing full details of the object while building the register, and express(), which determines how the object is represented in text. For instance, if Target is an object of class Person, then Target.express() might evaluate to “Victor March”, but Target.identity() shows much more information: the object name, the object grammatical attributes, and for each member the name, value and grammatic attributes.

Any class editor implemented for DEXTER should obtain the following information from the author.

1. For each class, the default gender of an object of the class.

\(^2\)When an object such as a text string represents something that could be a noun, whether that thing is something that can be counted or not helps determine the use of “the” versus “a” or “an”.
2. For each class, whether the gender is fixed or variable.

3. If the gender is variable, how the class should represent gender:
   
   (a) as a distinct value (say String).
   
   (b) by the gender attribute of a specific member, for example the gender of a person might be represented by the gender of their first name.

4. For each member of the class, the default gender of the member (which is often the default gender value of the class of the member.)

5. The default expression of the object (for the `express()` method.)

A dialog analogous to steps 1–4 above must be carried out for the number and person attributes. This information is tedious to enter by hand. It would be easier with a gui offering default values.

tutIKindAnimal. In Reg_Tutor, the `tutIKindAnimal` class is described as follows. Objects of the class have variable gender, with the default gender value being `UNCERTAIN`. Gender of an object is and represented by the gender of the object’s `IAnimalName` member. A `tutIKindAnimal` represents a single animal, hence the class’s objects are always `COUNT(able)` and `SINGULAR`.

The default string expression of an `tutIKindAnimal` object is a string of the form: the `IAnimalName (IKind )`.

For the `tutIKindAnimal` object with the values of “Carmen” (name) and “cat” (kind), this expression is “Carmen (cat)”.

The one-size-fits-all approach of the DexterObjectInterface seems clumsy in English. The gender of a Person class is variable, the gender of a furniture class fixed (NEUTER). In other languages gender is more important.
The tutIPof Class. I will not enumerate all of the details of the gender/count/number decisions for the tutIPof class. I will describe the intent of the members and point out interesting features.

TutIPof represents a person of focus. People are always countable and singular. Their gender defaults to UNCERTAIN, is variable, and is represented explicitly by the string ISex, which is intended to have the values M or F. The member strings IFName and ILName hold names like Victor and March respectively. Member strings IAge, IWeight and IHeight are meant to hold string descriptions, such as old, slim, tall. The SFName and SLName members hold the name of the current spouse, if any. The member IAnimalsOwned holds a possibly empty list of tutIKindAnimal, representing the person’s pets. Since it is a list, IAnimalsOwned is treated as a collective noun: SINGULAR and COUNT(able), of NEUTER gender. IAnimalsPofLikes is a list consisting of strings representing the kinds of animals the person likes, and IAnimalsSPofLikes lists the kinds of animals the spouse of the person likes. The default string expression of a person is just IFName ILName (for example Victor March.)

DEXTER supports aggregation of classes in the ontology (tutIPof aggregates tutIKindAnimal) but not inheritance among those classes.

Once the queries and classes are specified it is time to create the register with ADMIN.

4.2.2 Declaring a Register with ADMIN

ADMIN serves two purposes in DEXTER: creating registers (initializing various database tables, setting up directory structures and so on) and adding classes to registers (again updating various database tables, setting up directory structures, creating stub class files). If DEXTER had a functional query editor and a functional class editor there would be no need for ADMIN.

The author invokes ADMIN and at the prompt issues three commands:

1. create Reg_Tutor

2. add tutIKindAnimal Reg_Tutor

3. add tutIPof Reg_Tutor
The results of these actions are the creation of the register (first command) and inclusion of the two specified classes in the Reg_Tutor ontology. Numerous messages appear that I will not detail. ADMIN reports its actions and any errors, and creates a shell script to undo its actions.

Once the author creates the register and defines its classes it is time to implement the register field.

4.2.3 Building the Field with FRED

FRED is DEXTER's Field Rule EDitor. FRED allows the register author to define the objects of the internal scheme and the rules that map query results into those objects. FRED has facilities to view the form of query results, execute the queries and display the actual query results, and to view the classes of the ontology (which includes the DexterOntology), their members, and the set of Dexter built-in objects. Built-in objects include the command line, the name of the register, the current date, an error message, the system name of the user, and, for rules that handle more than one tuple in a query result, the number of rows, the current row, and the number of rows that satisfy the rule condition. The user may specify initial values for internal scheme objects or their members.

The author defines rules by specifying a rule name, the type of the rule (as in Section 3.5.2), and a query result on which the rule is defined. Once a rule is defined, the user tells FRED to generate a Java class implementing the rule and then to compile the class. Similarly, the internal scheme is generated as a Java class and compiled when its object list is modified. Once a rule is created, FRED allows the user to set runtime arguments and try the rule, either individually or in a list with other rules. The results may be displayed in a number of ways. The output of the express() method of each object in the internal scheme may be displayed, as may the more complete output of the identity() method. FRED also allows the author to define a fixed template into which the values of internal scheme objects may be substituted. This facility helps the author understand how the register constituents must be built and can serve as a notepad.

Choosing a train of thought. The author faces a choice when beginning to work with FRED. The author can define all the objects of the internal scheme and all the field rules as a first step. Doing so is consistent with the register-processing pipeline,
which begins with field processing and must complete field processing before proceeding further. An inexperienced register author might well be uncertain what objects and rules he needs, however, and an experienced register author may want to see if his plan, queries, ontology and so on, have any flaws that will become obvious in later processing. Another approach, the one our register author takes with Reg_Tutor, is to take a train of thought from the corpus document — one or two sentences for example — and carry that train of thought through the entire register to build just that portion of the document. A register is monotonic: the author can always add more objects and rules yet still compute the same values as previously. The only condition on monotonicity is that no new rule block a previously defined rule.

The train of thought the Reg_Tutor author chooses is the first sentence, which in paraphrase says that so-and-so is a [young to very old] [man or woman] who is [slim to obese], of height [short to tall]. For example: Victor March is an old, slightly overweight male of average height. The register author needs rules to verify that the person named on the command line exists in the database, and to find the age, weight and height descriptors, and the gender value for that person. The author's next task is to plan the field rules to get the data supporting this sentence.

Planning Rules. While planning the rules, the register author must notice:

1. to which query result a given rule applies;

2. whether the query can result in multiple tuples, no tuples, a single tuple;

3. if the query can result in multiple tuples, whether the rule should choose one tuple, iterate over all tuples, or use aggregate values determined from all tuples;

4. the internal scheme objects the rule will valuate;

5. whether the rule must reference internal scheme objects in its precondition, condition, or both.

The author's first task in FRED is to create an internal scheme by defining the necessary objects and then generating and compiling an internal-scheme class. A first field rule can then be built in FRED.
In Reg_Tutor, the first rule should verify that the name on the command line exists in the database. A reasonable action to take is to record whether the target person is found or not and to assign the name, when found, to an internal-scheme object. A pseudo-code sketch of the rule is shown in Example 4.1:

**Example 4.1** Rule sketch  // assume FoundTarget is initially FALSE

IF ( query getPerson returns exactly 1 row ) THEN
   FoundTarget is assigned TRUE
   Target.FName is assigned getPerson.FName
   Target.LName is assigned getperson.LName

The rule sketch in Example 4.1 indicates the following:

1. The internal scheme includes a Boolean value FoundTarget initialized to false. The rule sets FoundTarget true if the condition is satisfied and the name of the person can be assigned.

2. The getPerson query must return exactly 1 row. Otherwise the person requested does not exist (0 rows) or is multiply defined (more than 1 row).

3. The internal scheme needs an object Target of type tutIPof.

There is good reason why the register author chooses not to assign further values to Target's fields. Suppose this rule were to assign Target.ISex based on getPerson.Sex. The Sex attribute in the PEOPLE table may have the value null. To avoid problems later in the register4, the author would then want one rule with the extra conjunct getPerson.Sex is not NULL and another with the conjunct getPerson.Sex is NULL. The result would be a number of rules with more complex conditions, which is error prone. By not assigning more values the author does not decrease the number of rules, but he does decrease the complexity of each rule.

---

4Problems such as repeated checks for null values, or worse runtime problems.
Using FRED. The register author may now launch FRED to start creating the Reg_Tutor Field. Figure 4.2 shows a FRED screen. The menu bar gives access to all but a few actions. The rest of the screen is divided between two tabbed folders. The upper folder (meta folder) contains the lists of defined rules and queries on one tab, author-defined objects and all classes on another, built-in objects on a third, and the register-choice mechanism on the fourth. The lower folder (the working folder) has tabs for: rule assignments (consequent of the rule), rule condition and precondition (the nullcheck of Section 3.5.1), the results of query executions, various views of the internal scheme objects, messages from FRED, and an area to work with a fixed-template view of the internal scheme. Figure 4.2 shows the Dexter built-in objects in the meta folder, and the results of the getPerson query run on Victor March are shown in the working folder.

Figure 4.2: FRED, showing built-in objects in the meta folder, query results in the working folder.
Defining the first objects. To create the necessary objects for the first rule, the Reg.Tutor author takes the following steps, some of which are shown in Figures 4.3 through 4.5.

1. Make sure that the Reg.Tutor register is loaded (Register Selection tab in the meta folder.)
2. Select the Objects and Classes tab in the meta folder.
3. Click on the Define New Object button to get the Define Register Object dialog.
4. Enter FoundTarget in the Define Object text field.
5. Go to the Objects and Classes tab in the meta folder and choose the DexterBoolean class from Ontology Class Choice. DexterBoolean will appear in the Of Class text field in the Define Register Object dialog.
6. Leave the FoundTarget object as a scalar object by leaving No as the answer to the question Object is an array indexed by Ints in the Define Register Object dialog.
7. Click Accept in the Define Register Object Dialog. (Figure 4.3.)
8. Change the name in Define Object text field to “Target”.
9. Change the class name in the Ontology Choice to tutorial. The Of Class field changes correspondingly.
10. Click Accept. (Figure 4.4.)

The object FoundTarget should be initialized to false, which the author can guarantee by these steps:

1. Select FoundTarget in the Objects list.
2. Choose Edit Current Object Initialization from the OBJECTS menu.
3. Type SET = false in the text box in the Object Initialization dialog.
4. Click Accept (Figure 4.5).

The object list must now be saved by choosing the Save Current Object List option on the Objects menu. The next task is to define and build the first rule described above.
Figure 4.3: Defining FoundTarget.

The first rule. The first rule assigns the first and last names of Target and records the success if the query getPerson succeeds with just one row. To define and test this rule, the register author takes the following steps (partially captured as Figures 4.6 through 4.11)

1. Select the Rules and Queries tab in the meta folder.

2. Click the New Rule button.

3. Type the name of the rule, verifyTargetRule in the New Rule Name field. (Rule names must begin with a letter, can contain only letters, digits and underscores, and may not be prefixed by “get”, “set” or “is”.

4. Adjust the query choice to getPerson, the query on which the rule is defined.

5. Make sure the With Rule Type choice is singleton.

6. Click Accept (Figure 4.6). The rule will appear in the rule list in the Rules and Queries tab, and the text box in the Assignment tab of the working folder will change.
Figure 4.4: Defining Target.

A field rule in DEXTER may be one of five types, as described in Section 3.5.2.

**singleton rules.** Singleton rules fail if the result of the query on which the rule is defined does not have exactly one tuple. If the result has exactly one tuple, then the precondition and condition of the rule are evaluated on that tuple\(^6\).

**take-first rules.** Take-first rules handle results that can contain more than one tuple. Take-first rules apply the precondition and condition of the rule to each tuple in turn, until the first tuple satisfying the condition is found or until all tuples have failed. Once a tuple satisfies the conditions of a take-first rule, the rule does not examine any further tuples.

**iterative rules.** Iterative rules also handle multiple tuple results. Iterative rules are evaluated against each tuple in turn and act each time a tuple satisfies the condition.

Elements of an iterative rule's action list may be restricted to occur before any tuple

---

\(^6\)The restriction on name prefixing is a legacy of early development.

\(^6\)Preconditions check both internal-scheme values and query results for null values.
is evaluated (PRIOR actions), restricted to occur only when the first tuple satisfies the rule (INITIAL actions, which are not implemented in FRED), or restricted to occur only when all tuples have been evaluated (TERMINAL actions.) An unrestricted action occurs once for each tuple on which the rule succeeds. Like singleton rules, iterative rules fail on an empty result.

**aggregate rules.** Aggregate rules are defined to allow another way of handling multiple tuples in query results. (Section 3.5.2). Aggregate rules are defined but not implemented in DEXTER.

**introspective rules.** Introspective rules use only objects from the internal scheme in their conditions and actions. They do not directly reference query results, but must be defined with respect to some query.\(^7\) Introspective rules are convenient. The author

\(^7\)I have preserved the option of requiring that an introspective rule require prior evaluation of a given
can use Boolean variables to record that previous rules did or did not succeed, and take actions accordingly. This technique allows one rule to fire only if another did (did not) succeed, or even based on the success of sets of rules.

Any field rule in DEXTER has the form:

**IF ( (nullcheck) AND Condition ) THEN**

**Action List**

The nullcheck (or *Precondition*) of the rule verifies that query-result columns used in the rule are null or non-null, as appropriate to the rule. For instance, since query, or even of all non-introspective rules defined on the same query, even though it will not directly access the results. I have not yet needed to implement that requirement.
DexterObjectInterface allows assignment of the value Empty to an object, it is usually a mistake to assign an object a null value read from a result column. If the Precondition is satisfied, the Condition is evaluated. Condition is a general Boolean condition on the attribute values of the current tuple and/or the values of the objects of the internal scheme. If Condition is satisfied, then the Action List is carried out. Possible actions are: assignments to the objects of the internal scheme, invocations of object methods, executing verbatim Java code and running an external program. Executing verbatim Java code or executing an external program allow authors who know how to program to bypass DEXTER limitations. Section 5.3, describing the Pulsar registers, provides an example.

The Precondition and Condition of verifyTargetRule may be left empty. Specifying that the rule is a singleton guarantees that the result has exactly one generic tuple. That tuple must have LName and FName attributes match in the arguments register arguments. Those names may be stored directly in targetIFn and Target.IILName. The verifyTargetRule also assigns FoundTarget true. Once those assignments are made, DEXTER requires the author to indicate the object assignment is complete by setting TargetIFn, Target.IILName and FoundTarget ASSIGNED. Since the objects of the internal scheme are often composite, the DexterObjectInterface requires that an object be marked when its assignment is complete. Making this assignment of assignment is tedious, and a production version of DEXTER would behave differently in this respect. The full assignment of the verifyTargetRule along with FRED's edit-tools palette is shown in Figure 4.7.

To create this assignment, the author carries out these steps:

1. Open the Assignment tab in the working folder. Use the mouse to cut out any initial text that appears there\footnote{A relic of development.}.

2. Use the Edit menu's Show Edit Tools selection to show the editing-tools palette, which allows the author to access internal scheme objects and their members correctly and without typing.

3. Select the Objects and Classes tab in the meta folder.

4. Select FoundTarget from the Objects list.
Figure 4.7: The assignment of verifyTargetRule, showing FRED’s edit tools palette.

5. Type `SET` on the first line of the `Assignment` text field.

6. Click `OBJECT` on the edit-tools palette, placing `FoundTarget` after `SET`.

7. Type `TRUE` after `FoundTarget`.

8. Open a new line in the text box on the `Assignment` tab, and type `SET` on that line.

9. Choose `Target` from the `Objects` list in the `Objects and Classes` tab of the meta folder.

10. Find the `tutIPOf` class in the `Ontology Class` choice on the same tab.

11. Select the `IFnName` member of the `tutIPOf` class in the list describing that class (still on the `Objects and Classes` tab in the meta folder.)

12. Click `OBJECT` and then `.mem/meth()/const` on the editing-tools palette. The current line of the text box in the `Assignment` tab now changes to `SET Target IFnName`. The author adds a trailing space.
13. Select the Rules and Queries tab in the meta folder.

14. Move the getPerson query.

15. Select the FName column of the getPerson query.

16. Click COLUMN on the editing-tools palette. The current line of the assignment now reads “SET Target.IFName getPerson:FName”.

17. Create the assignment of Target.ILName by changing to the Objects and Classes tab, and selecting the ILName field of the tutIPOf class, and then using the editing-palette buttons OBJECT, .mem/meth(/)/const and COLUMN again.

18. Open the Edit menu and use the mouse to copy and paste the three assignment lines, making six lines in the assignment.

19. On each new line, change the value assigned to the object to “ASSIGNED”.

The Assignment tab’s text box allows typing. Though tedious to describe here, the gui is actually faster and more reliable to use.

**Testing the first rule in FRED.** At this point the first rule is defined. The author would like to use FRED to test the rule. The author must take two steps prior to testing verifyTargetRule. He must build the internal scheme and build verifyTargetRule. Building the internal scheme is necessary, because FRED relies on the runtime internal-scheme class to build the field-rule classes. FRED is a Java application compiled before the ontology classes and internal-scheme class of any particular register exist. Generation relies on java.lang.reflect to determine the validity of internal-scheme references within a rule. Java’s reflection API allows a string to be turned into a class object that can be interrogated for member names and types, used to create an object of the class, and so on.

To build the internal scheme and the verifyTargetRule, the author does the following:

1. Choose Generate Internal Scheme from the BUILD menu shown in Figure 4.8. This action translates the saved object list into a Java class

2. If generation is successful, return to the BUILD menu to choose Compile Internal Scheme. This action compiles the InternalScheme class. Again, messages appear in the Messages tab.

3. Return to the BUILD menu to choose Generate Rule and then, if messages indicate successful generation, Compile Rule.

![Run time Args menu]

Figure 4.8: FRED’s BUILD Menu.

Now the author can test the rule and see what happens in the internal scheme (that is, verify that the correct assignments have been made to the `FoundTarget` and `Target` objects.) To do so the author must:

1. Return to the BUILD menu and select the Set Runtime Args option from the Run time Args ... submenu.

2. Type Victor March in the text field and click ACCEPT. (FRED echoes the arguments in the FYI dialog.) See Figure 4.9.

3. Return to the BUILD menu and choose Execute this Rule. The FYI dialog will notify the author when the execution is completed, and the author may view messages in the Messages tab of the working folder. If the rule succeeds, the message will be (1) `verifyTargetRule = true` No Message. In the event of failure of a rule, FRED tries to say why. For example, the Precondition might not have been satisfied (if a result
column was found null in a rule that required the column), or one of the conjuncts in the condition might have failed.

![Figure 4.9: Simulating a register's run time arguments in FRED.](image)

**Viewing Internal Scheme Objects.** One way to view the state of the internal scheme objects is to go to the *Int Scheme View* tab on the working folder. If the *View As* choice is *expression*, then choosing *View All Objects* shows the author the result of the *express()* method of each object. If the *View As* choice is *All Details*, the author can see the internal state of the objects (the *identity()* method output). See Figure 4.10 for the detailed view of both *FoundTarget* and *Target* (with the *Target.IName* member object highlighted.)

Another way to view the internal scheme objects is in a fixed template. FRED allows the author to define, save and retrieve a fixed template with references to internal scheme objects. An example of using this facility is as follows:

1. Select the Template View in the working folder.

2. Enter (for example) *About* in the text box by typing in it.
Figure 4.10: Detailed view of two objects after `verifyTargetRule` is executed on Victor March.

3. Click *Show References* to obtain the valid internal scheme references in the list on the right of the tab.

4. Select a reference, for example *Target*, and click *Insert Reference*. The string "Target" appears at the cursor in the text box. See Figure 4.11.

5. Click *Fill*. The references are replaced with the value obtained by invoking the `express()` method of the *Target* object. In this case, an object of type `tutIPof` outputs the `IFName` member, a space, and the `ILName` value resulting in *About Victor March*.

Multiple templates can be defined and saved. The templating facility is useful for several purposes. A fixed template allows the author to keep track of what he wants to do and allows him to see where the functionality of the register must exceed a simple template. A template can also serve as the author’s notepad.

**More field rules.** The preceding discussion has covered writing the first field rule in FRED. Although the details are numerous when written down, once the author understands a rule, it takes only minutes with FRED to write and test the rule. The rule is constructed as a Java class, which is compiled before the register is executed. On execution the register constructs an object of the class. The rule is evaluated as a side effect of object construction. The class for this particular, simple rule contains 52 executable Java statements. Rather
Figure 4.11: FRED’s fixed templating facility.

than detailing each of the successive rules in FRED, I describe the rules in English and
discuss implementing them in FRED only when there is a specific point. Ten more field
rules provide complete field support for the first sentence, specialized to someone like Victor
March.

MissedTargetRule complements verifyTargetRule. VerifyTargetRule sets
FoundTarget, initially false, true. Therefore, if FoundTarget remains false after
verifyTargetRule is evaluated, the name on the command line was not found in the
database.

Example 4.2 MissedTargetRule (pseudo-code):

IF ( true ) and (not FoundTarget)) THEN
    set ErrorMessage indicating command line name not found

MissedTargetRule is an introspective field rule. It does not access any query result.
Hence the Precondition is empty. Introspective rules are not strictly necessary, but they
are convenient, particularly when coupled with epistemic objects such as FoundTarget. I
refer to Boolean objects as epistemic objects because they serve to inform the register what
it knows as it processes. DEXTER’s register implementation encourages use of epistemic
objects; once structure processing begins, all access to the internal scheme must be through
epistemic objects. Order of rule evaluation in field rules is almost always important. In this case the author must check verifyTargetRule before checking missedTargetRule. The next rule is called determineSexRule. First the author associates a new epistemic variable, SEX_DEFINED, with Target and initializes it to false. The rule itself appears as follows in FRED's View Current Rule dialog:

**Example 4.3 determineSexRule**

IF ( [ getPerson:Sex IS_NOT_NULL ] And [ FoundTarget ] ) THEN
  SET Target.ISex getPerson.Sex
  INVOKE Target.ISex.setGram_gender( getPerson:Sex )
  SET SEX_DEFINED TRUE
  SET Target.ISex ASSIGNED
  SET SEX_DEFINED ASSIGNED

**DetermineSexRule** is a singleton rule defined on the getPerson query. The Precondition checks that query getPerson does not return a null in its Sex column. Assigning a null value to Target.ISex could cause runtime problems later in processing. DEXTER supports a notion of assigned empty, which would be the preferred action if getPerson.Sex were null. The Condition only checks that FoundTarget is true, indicating that it makes sense to try to determine the sex of the target. The action list contains one new wrinkle: the invocation of the method Target.ISex.setGram_gender. The author chooses to represent sex explicitly in tutIPof with member ISex. There is nothing wrong with representing sex explicitly, but it turns out to not be very useful in DEXTER. In the register's tenor, DEXTER expressions cannot check if Target.ISex is M or F, nor if it is Male or Female. To later allow expressions to determine the gender of Target, the author also sets the gender of Target.ISex using the DexterObjectInterface method setGram_gender. The setGram_gender method can be parameterized with either an integer (DexterObjectInterface defines MASCULINE as the integer 7) or with a string. The method automatically recognizes M or m as MASCULINE and F or f as FEMININE. Male, for instance, is not recognized. Had the database recorded gender as other than M and F, then the determineSexRule would need to be split to deal with, for example, getPerson:Sex = "male".

The determineSexRule fails if the data for Target has a null in getPerson.Sex. Therefore, the author should use another rule, determineSexEmptyRule, as follows.
Example 4.4 determineSexEmptyRule

IF ( [ true ] AND [ FoundTarget And ! SEX_DEFINED ] ) THEN
  SET Target.ISex EMPTY
  SET Target.ISex ASSIGNED
  SET SEX_DEFINED ASSIGNED

DetermineSexEmptyRule is another introspective rule. In this rule, the author checks epistemic information: if the field knows that it has found the person requested (FoundTarget is true) and even after determineSexRule has been evaluated, SEX_DEFINED remains false, then the data cannot give the sex of Target. The rule assigns Target.ISex the value empty and marks SEX_DEFINED assigned (false) as well. The author need not include the line INVOKE Target.ISex.setGram_gender( tutIPof.UNCERTAIN ), because Target.ISex defaults to UNCERTAIN gender. DetermineSexEmptyRule must be evaluated after determineSexRule.

The author could avoid determineSexRule by assigning Target.ISex at the same time he assigned FoundTarget. However, verifyTargetRule would then have to include the Precondition getPerson.Sex is not null. For each of \( n \) attributes the author might assign in verifyTargetRule, the author would have to account for the possibility of a missing attribute value. Missing attribute values can only be noticed in the condition of a rule. Since a rule to handle each missing attribute is necessary in any case, the author elects to keep verifyTargetRule simple.

The next piece of information the author seeks is the age of the Target. He decides that Target.IAge should be a string descriptor of age, such as “old” or “young”, rather than a numeric value. The rules that assign values to Target.IAge transform numeric data into string data. Since the author is only trying to support the first sentence specialized to “Victor March” at this time, he only needs one rule: determineAgeRule_4. (The numeric suffix serves to remind him that there will be a family of age rules.) Following his habit of using epistemic objects, the author creates a Boolean internal scheme object AGE_DEFINED and initializes it to false. The rule appears as follows.

Example 4.5 determineAgeRule_4
IF ( [ getPerson:Age IS NOT NULL ] AND
    [ FoundTarget And 51 <= getPerson:Age And getPerson:Age <= 70 ] )
THEN
    SET AGE_DEFINED TRUE
    SET Target.IAge "old"
    SET AGE_DEFINED_ASSIGNED
    SET Target.IAge ASSIGNED

DetermineAgeRule is a straightforward singleton rule defined on getPerson. In the text box of FRED’s Condition and Precondition tabs, each line represents a separate conjunct (Figure 4.12.) By choosing to assign Target.IAge the value “old”, the author makes a grammatical commitment. Eventually the string value of Target.IAge will appear in the final document. The expressions that use this value must accommodate “old” grammatically. For example, “He is an old man” works, as does “He is old.” Terms for other age groups must be grammatically compatible. For instance, “young” works (grammatically) most places “old” works. (“He has entered {old,young} age” would be an exception.) “Middle age”, however, will not do: “He is a middle age man.” It must be “middle aged”, which is still not perfect. For instance, “He is of old age” and “He is of young age” are (barely) passable, while “He is of middle aged age” fails. Use of string values in the field requires care.

If DEXTER implemented a vocabulary management component, then “old” would be defined in that vocabulary as indicating a person between 51 and 70 years of age in the Reg_Tutor register. If the vocabulary management were sophisticated, perhaps along the lines of wordnet [18, 50], then it might be possible to allow expressions to look up root words and choose “middle aged” rather than simply using “middle age.” So vocabulary management might make string values easier to use. DEXTER, although relying on templates for expression, can manage to say “an old” or “a young” as needed, without benefit of a vocabulary component. That ability relies on built-in procedures that use linguistic knowledge and the grammatical attributes of the DexterObjectInterface.

As with Target.ISex, there is a rule to set Target.IAge empty should there be an unrecognized age (such as null or −22) in the data. AssignAgeEmptyRule is completely analogous to determineSexEmptyRule. FRED recognizes that rules may be quite similar
and allows for definition of a new rule by copying ("inheriting") another rule’s Precondition, Condition and Assignment. (The option is visible, but disabled, in in the dialog of Figure 4.6.)

![Diagram of FRED interface](image)

**Figure 4.12:** Conjuncts appear one per line.

The author uses two sets of rules and an epistemic object to determine the string value describing age. The first set of rules contains the rules that assigns “old”, eventually “young”, and so on. **DetermineAgeRule** is one member of the set. The second set contains only the rule to mark age as empty or unknown. The sex attribute is handled similarly, as is height. For weight, things are a different. The author intends to rely on both the weight and height stored in the database to compute the BMI (Body Mass Index\(^9\)) of **Target** and from that value assign a string-valued age descriptor to **Target.Weight**. Thus the author uses an epistemic object, **BMI.DEFINED**, as well as a separate numeric object, **Target.BMI**, to hold the computed BMI. He also uses three sets of rules. The first set contains one rule that computes and stores the value of **Target.BMI** and sets

---

\(^9\)BMI is the ratio of weight in kg to height in cm (squared).
**BMI.DEFINED** true. This is a singleton rule, requiring exactly one tuple in the result and defined on the query **getPerson**. The second rule set contains introspective rules, each conditioned on **BMI.DEFINED** being true and the value of **Target.BMI**; each assigns an appropriate string value to **Target.IWeight**. The third set again has only one rule that will mark **Target.IWeight** empty if **BMI.DEFINED** is false. Again, rule order matters, though not within the second set of rules. **DetermineWeightRule** is another singleton rule defined on **getPerson**:

**Example 4.6 determineWeightRule**

```
IF ( [ getPerson:HtIn IS_NOT_NULL And getPerson:WtLb IS_NOT_NULL ] AND
    [ FoundTarget ] )
THEN
  SET BMI.DEFINED TRUE
  SET Target.BMI (( getPerson:WtLb * 0.45) / )
    (( getPerson:HtIn*0.0254)*( getPerson:HtIn * 0.0254))
  SET BMI.DEFINED ASSIGNED
  SET Target.BMI ASSIGNED
```

Specializing the first sentence to **Victor March** the author needs only the rule to handle the associated BMI. This is the introspective rule **assignWeightRule_2**.

**Example 4.7 assignWeightRule_2**

```
IF ( [ true ] AND
    [ FoundTarget And Target.BMI >= 24 And Target.BMI < 27 ] )
THEN
  SET Target.IWeight "slightly overweight"
  SET Target.IWeight ASSIGNED
```

□
This final weight rule, assignEmptyWeightRule, is analogous to the other “Empty” rules. Height is the only attribute left to deal with. The author describes height differently for men and women. The distinction eventually doubles the number of height rules. (That number would increase if the author chose to also distinguish between adults and children, but he knows his data only pertains to adults.) Specializing to Victor March, the author only worries about one singleton height rule for men defined on getPerson, and the usual introspective empty rule (assignHeightEmpty, not shown). As expected, the author also defines the epistemic variable HEIGHT_DEFINED. FindHeightRule.M2 appears as follows.

**Example 4.8 findHeightRule**

```
IF ( [ getPerson:HtIn IS_NOT_NULL ] AND
     [ FoundTarget And getPerson:Sex = "M" And getPerson:HtIn >= 67 And
       getPerson:HtIn <= 71 ] )
THEN
  SET HEIGHT_DEFINED TRUE
  SET Target.IHeight ‘‘of average height’’
  SET Target.IHeight ASSIGNED
  SET HEIGHT_DEFINED ASSIGNED
```

The value “of average height” is necessary for an average-height individual as the author intends to use “tall” and “short” to express “He is tall” and “He is short.” “Of average height” is grammatically compatible in most cases, whereas “average” is not.

Implementing these rules in FRED, the author develops full support for the first sentence of his planned document for anyone who is “slightly overweight, male, and of average height.” Each rule is tested individually in FRED as the rule is built, although doing so requires commenting out the FoundTarget conjunct appearing in almost all conditions subsequent to verifyTargetRule. The author also puts the rules in a list and execute them directly in FRED. Figure 4.13 shows the Execution List … dialog and the working folder’s Messages tab just after the rules described here were executed on the data given above, with the runtime arguments “Victor” “March”. The author gains access to the Execution List … dialog through the BUILD menu. By selecting rules in meta folder’s
Rules and Queries tab and the controls in the dialog, the author can build and save an ordered list of rules for execution. That list determines the order of execution for the rules. The dialog informs the author if runtime arguments are or are not defined. The message window shows the result of each rule execution. The expected rule executions return true with no message, as exemplified by (1) `verifyTargetRule result = true` No Message. The expected rules also fail, for example: (2) `missedTargetRule = true` Message: Condition Failed. The result of `missedTargetRule` is true because the rule could be evaluated. A result of false indicates that something prevents the rule’s evaluation, for instance the InternalScheme class cannot be found. The message that the condition fails indicates that the rule does not succeed when evaluated. Figure 4.14 shows the results expressed in a saved, fixed template called `sentence_one` created in FRED. The author has made a note that this is the best he can do with these rules and a fixed template.

![Figure 4.13: Executing a list of field rules in FRED.](image)

4.2.4 Determining content: CONED, CODEX, MORCONED

The register now has the data represented to support the first planned sentence, at least for an individual resembling “Victor March.” The author must now tell the register that there is such a sentence by creating a content-message constituent. Since the author has decided
Figure 4.14: Reg_Tutor’s sentence_one fixed template filled after list execution.

to handle the case that the Target cannot be found, he should also define constituent to handle error messages. Content messages are created in CONED, DEXTER’s CONstituent EDitor.

CONED

In the current example, the steps to create the two needed content messages are as follows:

1. Invoke CONED.

2. Use the Current Register choice to load Reg_Tutor.

3. Click Add Message to get the Add Content Message dialog.

4. Give the message a name by typing in the New Message Name text field. For example, IDAgeHtWt.

5. Click Add Parameter to get the Add Parameter Dialog.
6. Type a name for the parameter in the Parameter Name text field. For example, Person.

7. Choose whether the parameter may or may not be passed as null. In this case, null is not allowed. Alternative expressions for the message could be used to account for a null Person, but because the author uses FoundTarget and a separate rule for the case that the requested person (Target) cannot be found, it makes sense to use a separate message for errors.

8. Choose whether the parameter represents an array or a scalar. Person is a scalar.

9. Choose a type for the parameter. Person should be a tutIPof object.

10. Click Accept to attach the parameter to the message. Figure 4.15 shows the state of the editor just after Accept is clicked, defining Person as parameter that is a tutIPof scalar that cannot be null. The author could have chosen to define parameters representing the pass the individual fields of a tutIPof object rather than an entire tutIPof object.

11. Click Close in the Add Parameter dialog.

12. Decide if the message should be repeatable or not. The IDAgeHeight message should not be repeated within the Reg-Tutor document. The Comment text field can hold an arbitrary comment describing the intended use of the message. See Figure 4.16.

13. Click Accept Message in the Add Message dialog.

14. Click Build Current CM to build a class representing the constituent.

15. Repeat steps 2 through 13 to add the repeatable ErrorMessage constituent, which accepts one possibly null string parameter msg.

Figure 4.17 shows CONED with both messages added. A constituent in DEXTER is essentially an expressive function. Specifying a constituent in CONED is analogous to declaring the function's prototype in C or C++. The function definition is the set of expressions associated with the constituent via CODEX and interpreted when the register is executed.
Figure 4.15: CONED defining IDAgeHtWt with parameter Target in Reg_Tutor.

Now the author faces a choice of direction. He can immediately define the expressions associated with the constituents just created by using CODEX, or he can defer doing so until as late as after defining the document structure in STRUCTED. In the latter case his next step would be to use MORCONED to write the rules that will either bind the internal scheme’s Target object to IDAgeHtWt’s Person parameter, or bind the built-in ErrorMessage object to the just defined ErrorMessage constituent’s msg parameter. In this case the author chooses to move to CODEX to give at least one expression to each message.
Figure 4.16: CONED ready to accept message \texttt{IDAgeHtWt}.

**CODEX**

CODEX is the COnstituent Data EXpression editor. CODEX is used to define a set of possible expressions associated with a constituent created with CONED or PRESED. When the execution engine is ready to include a message in the final document, it finds the expression definitions associated with the message, selects an available expression definition, and interprets that definition. The interpreted definition is the expression of the message in the document. In the Reg.Tutor register the author has two messages that need expression: \texttt{IDAgeHtWt} and \texttt{ErrorMessage}. The author begins as usual by launching CODEX and making sure Reg.Tutor is the active register. CODEX resembles FRED in that there is an upper tabbed folder of meta information and a lower tabbed folder of working space. Since \texttt{ErrorMessage} has no expressions defined, the author may toggle between the upper tabs \textit{Expressions of This Message} (which is empty) and \textit{Message Description}, shown in Figure 4.18.

The message description has two core components. Each parameter is listed along with its
type in the Parameter Components list. If the parameter is a composition, then the members, for example Target.IDName, are shown with their types. The aggregation hierarchy is represented by indentation. Focus settings are also shown in the message description. CODEX currently defines two foci, SUBJECT and OBJECT. The list of foci may be modified by editing the CodexProperties file. Foci are discussed in Sections 2.3.2 and 3.3.

Initially the author plans two expressions for the Error Message constituent. The first warns that an error occurred and then show the value stored in msg. The second covers the case that the Error Message constituent needs expression when msg is null. The author assumes HTML as the representation language. He begins with the message when msg is null, and first specifies in what contexts and under what conditions this expression may be used. He takes the following steps:

1. Choose New Expression from the Expressions sub menu of the File menu. The lower tabbed folder becomes active.
Figure 4.18: CODEX showing Message Description tab.

2. Determine the allowed contexts of the expression. In the Contexts tab the author may choose to allow the expression only in particular contexts, only in contexts of a particular kind, or in all contexts. In this case, he allows all contexts by leaving the Allowed Context(s/Types) list empty.

3. Determine the requirements (other than context) on the expression. Which parameters are required to be present or absent? Which foci? In this case the author requires only that the msg parameter be null. The steps to indicate that requirement are:

(a) Select msg in Message Description tab.
(b) Click Add Selected Parameter to List in the Requirements tab.
(c) Select the entry “msg_is_Present”, which appears in the Current Expression Availability Depends On list in the Requirements tab.
(d) Click require EXcluded, changing entry to “msg_is_Missing”.

Defining Availability of the Expression. Expression availability is determined by the constraints the author places on context, constituent parameters and foci when defining
the expression. Contexts and foci are described in Sections 2.3.2 and 3.3. DEXTER recognizes five kinds of contexts. If a constituent has Title in its name, then it is a title context. The same applies to the terms heading, paragraph, lead and conclusion. A single context may be an instance of any, all, or none of these context types. For instance, a message ConclusionParagraph is both a conclusion context and a paragraph context. An expression may be defined as available only in one or more specific contexts, or in one or more kinds of contexts, such as both titles and headings, or in all contexts.

Context alone need not determine availability. A parameter is defined as missing if it is empty or unassigned (in the DexterObjectInterface sense) or null. Otherwise the parameter is present. A focus is present when its value is set, missing otherwise. An expression may be available only when certain parameters and or foci are present and others missing.

Finally, an expression is available if and only if it is allowed in some open context, and all parameters required present (missing) are present (missing) and all foci required present (missing) are present (missing). One of the available expressions is selected at random in DEXTER. Thus a single constituent may have a family of associated expressions that can handle a variety of situations. Using multiple messages for a single expression is sometimes preferable to using several messages, each with rules to determine whether it is the usable version for the current data. The capacity to allow multiple expressions of the same constituent under varying data conditions is a significant part of DEXTER’s “flexible templating” approach.

**The Expression.** In the ErrorMessage case the message is canned text. The author moves to the Current Expression tab and enters this text in the text box:

\n<!-- ErrorMessage case msg null -->\n
An <b>ERROR</b> occurred, but there was no message attached.\n
DEXTER supports plain text and HTML output. Plain text requires post-processing with a pretty-printer. The default is HTML. The canned text makes use of various common programming conventions, such as using C-style new line characters for explicit line breaks.

After entering the above text, the author moves to the File menu, the Expressions sub-menu, and chooses Save Current Expression to save the expression in memory. To save the modified expression list to disk the author then chooses the File menu’s Save Work option. A saved expression is identified by an index, target, list of required contexts (empty again
indicating all), its requirements, and a prefix of its text. For the expression just above that identification appears as follows.

(1) {HTML}{}

The author embeds HTML comments indicating the name of the message and a description of the expression case. These comments could be generated as a semantic tag. (See Section 2.4.1.)

With this much background it will be easy to follow the author as he creates an expression for IDAgeHtWt. The steps are:

1. Select the IDAgeHtWt constituent and open a new expression.

2. Allow any paragraph context: on the Contexts tab highlight Paragraph in the Context Classes list and click Allow Selections, causing *Paragraph* to appear in the Allowed Context(s/Types) list. Leave the Use Context Classes checked.

3. Move to the Requirements tab. In the Message Description tab select each of Person.ILName, Person.IFName, Person.ISex, Person.IHeight, Person.IAge and Person.IWeight. In each case, click Add Selected Parameter to List, causing each to be required present. In this case the author is correct to require these six members of the Person parameter, rather than the Person object itself. The expression uses only these six members. Further, the actual parameter is Target, and at present the field rules do not set Target assigned. Person therefore will not be present formally, although these six attributes will be present.

4. Go to the Current Expression tab and enter the following text:

```
\n
<!-- IDAgeHtWt case IFName ILName IAge ISex IWeight IHeight -->\n```

5. Select Person.IFName from the Message Description tab and and click Insert Parameter Component on the Current Expression tab. This action creates a reference to the Person.IFName member just as in FRED's fixed template tool. At runtime this value will be substituted with the appropriate value.

6. Insert a space and a reference to Person.IName then type a space and is.
7. Go to the WINOOGS menu and choose Show Procedures. When the Procedures window appears select article.

8. Select Person.Age from the Parameter Components list.

9. Go to the ACTIONS menu of the Procedures window and choose Apply Procedure to Parameter. The following text will appear in the current expression: article(Person.Age). The text invokes a built-in procedure that determines whether “a”, “an” or “the” should be the determiner for the string in Person.Age. The procedure is heuristic and usually correct.

10. Insert a space and a reference to Person.Age followed by \n and a new line.

11. Go to the WINOOGS menu and choose Show Choices. The DOI Choices window will appear.

12. Select “Gender” in the left hand list of the DOI Choices menu. Possible values for gender appear in the right hand list. These values are only for reference.

13. Select Person.ISex in the Parameter Components list on the Message Description tab.

14. Click Apply Choice to Selected Parameter in the DOI Choices window. The result is pictured in Figure 4.19.

15. Hide the DOI Choices window.

16. Type appropriate text into the \{} following each occurrence of respond: For example, after “Masculine” type man inside the braces.

17. After the terminal ‘\}’ of the gender choice type who is, followed by a reference to Person.IHeight, the text and, and a final reference to Person.IWeight. Necessary spaces must be typed.

18. With the lower-right-hand choice of “.”, “?” and so on set to a period, check Terminator. This forces the expression to be terminated with a period.

19. Save the expression and the altered expression list. (File:Expressions:Save Current Expression and File:Save Work.)
The final expression text should appear as in Example 4.9.

**Example 4.9 IdAgeHtWt expression**

\n
<!-- IDAgeHtWt case IFName, ILName, IAge, ISex, IHeight, IWeight -->\n{Person.IFName} {Person.ILName} is {_article({Person.IAge})} {Person.IAge}\n{
  "evaluate {Person.ISex} for {Gender}"
    "finding {Masculine} respond: { man }"
    "finding {Feminine} respond: { woman }"
    "finding {Neuter} respond: { person }\n        ERROR: GENDER NEUTER --> }
    "finding {Uncertain} respond: { person }"
}
who is {Person.IHeight} and {Person.IWeight}

As a final act in this expression, the register author chooses to have this expression set the register's SUBJECT focus. Since the focus must be a DexterString, the author chooses Person.IFName as the focus. He selects this parameter component and the SUBJECT focus, and clicks *Parameter Sets Focus* in the *Message Description* tab. He will therefore want to go back to the field rule determineSexRule and make sure that he sets the gender of Target.IFName. This combination of actions will allow the next sentence to recognize that the SUBJECT is both known and masculine, enabling that message to use a masculine pronoun. In this small register, it would be safe to simply write the expressions of the second sentence to check the gender of the person, but in a more complex register the current subject may not be known at compile time. The data may force a rearrangement of the structure of the document. An expression may only set a focus, however, if the focus is not already set. Section 4.2.5, in the discussion on structure rule actions, describes how a register author indicates that a focus should be set.

One strength of registers and DEXTER is that they can define multiple equivalent expressions. For instance, here is an alternative expression semantically equivalent to the previous expression.
Figure 4.19: CODEX showing gender choice on Person.IAge.

**Example 4.10** An alternative expression

\n
<!-- IDAgerHtWt case IFName, ILName, IWeight, IAge, ISex, IHeight -->

{Person.IFName} {Person.ILName} is _article({Person.IWeight}) {Person.IWeight},

{Person.IAge}

{evaluate {Person.ISex} for {Gender}}

     "finding {Masculine} respond: { fellow }
"finding {Feminine} respond: { gal }
"finding {Neuter} respond: { individual <!-- ERROR: GENDER NEUTER --> }
"finding {Uncertain} respond: { individual }
}
who is {Person.IHeight}

These alternative expressions provide for the variants: Victor March is an old man of average height and slightly overweight and Victor March is a slightly overweight, old fellow of average height. It is up to the register author to ensure that alternative expressions are equivalent in both content and tone. If the author wants to vary the order of attribute expression based on value, for instance stating weight before all but name only if weight is extreme, a separate message, rather than separate expression, is required.

Finally, suppose the data does not include anything but Mr. March’s name and age. Then the Reg.Tutor author could define a similar expression for IDAgeHtWt requiring only Person.IFName, Person.ILName and requiring Person.IAge present, and Person.ISex, Person.IHeight and Person.IWeight missing. The missing requirement would be critical; otherwise the abbreviated expression would be applicable, and could be chosen, even when all information was present. (The register would be randomly withholding information, thereby failing the communicative goals set forth by Goffman (Section 1.4.4.) The abbreviated expression might look like this:

**Example 4.11** Abbreviated expression

```

<-- IDAgeHtWt case IFName, ILName, IAge -->
{
{Person.IFName} {Person.ILName} is _article({Person.IAge})
{Person.IAge} person

```

Unfortunately the author is denied the gratification of seeing his expressions rendered immediately. He still has to use MORCONED to write the rules that bind an argument to IDAgeHtWt, and still has to use STRUCTED to write the rules to place that message in an outline. Further, since he requires that IDAgeHtWt be expressed in a paragraph context
(and it would not fit in a title), he must define some paragraph context (using PRESED) and give it expression (using CODEX again).

**MORCONED**

Once the author creates the internal scheme and the rules to instantiate it using FRED, and then uses CONED and CODEX to define messages and expressions, which, when parameterized with internal scheme objects, express the data in a desirable way, his next task is to define the register content rules. The register content rules, created with MORCONED, bind internal scheme objects to message parameters. MORCONED stands for MOde Rule CONtent EDitor.

The form of a DEXTER content rule follows:

```
IF ( (Standard Predicate List) AND Condition ) THEN
    Action List
```

The *Standard Predicate List* constitutes the precondition of the rule, and consists of four predicates. Each predicate is defined on any object of a class implementing the DexterObject Interface. Because all classes of the register ontology implement the DexterObject Interface, the predicates are defined for all objects of the internal scheme. The four predicates are *Know*, *Don't Know*, *Know Empty* and *Know Value*. *Know(X)* is true if and only if X is an object for which the DexterObject Interface method *isAssigned()* returns true. The method invocation *X.isAssigned()* returns true if the action *SET X ASSIGNED* was taken during field rule evaluation. *Don't Know* is the complement of *Know*. *Know Empty(X)* is true if and only if *Know(x)* is true and the DexterObject Interface method *isEmpty()* returns true. The method invocation *X.isEmpty()* returns true if the action *SET X EMPTY* was taken during field rule evaluation. *Know Value(X)* is true if and only if *Know(x)* is true and the *isEmpty()* method returns false. The Standard Predicate List referred to in the form of a mode content rule is a conjunction of these predicates applied to selected objects of the internal scheme. *Condition* is a general Boolean condition on the objects of the internal scheme, although in practice conditions usually turn out to be simple conjunctions of numeric comparisons and checks on epistemic objects. Condition is evaluated only after the Standard Predicate List. The condition of a content rule has a strong epistemic flavor; content rules determine what can be stated in the document based on what
is known. The Action List of a content is list of actions limited to execution of verbatim Java code and asserting a content message with specific arguments. The arguments are either internal scheme objects, null, or literal values. Assertion of a message causes the message, with computed arguments, to appear in the set of messages forming the output of content processing and the input of structure processing. MORCONED does not check type compatibility between formal and actual parameters, but it does prevent assignment of an explicit null argument if a parameter is not allowed to be passed the value null.

Planning Content Rule in Reg-Tutor. The author needs two rules to support his defined messages. FirstIdentification is a rule supporting the first planned sentence. If possible, it will binds the internal scheme’s Target object to the IDAgeHtWt content message. Since the author has defined the ErrorMessage constituent, he needs to arrange for it to be selected in case the requested person is not found. The rule to select ErrorMessage, NoneSuch, tries to pass the built-in ErrorMessage object as the argument to the ErrorMessage constituent. The rules are described as follows;

Example 4.12 Rule descriptions

// FirstIdentification
IF ( Know( Target.IFName, Target.ILName, Target.IAge, Target.ISex,
             Target.Weight, Target.IHeight ) AND (true) )

THEN
    assert IDAgeHtWt( Target )

    and

// NoneSuch
IF ( not FoundTarget )
THEN assert ErrorMessage( _ErrorMessage )

In the first rule, the author checks that the register knows individual fields of Target, rather than Target itself. The current set of field rules does not mark Target assigned, so formally Target is not known in the register. However, as the field rules assign values to the members of Target, those members are marked assigned. The author does not check that the register knows values, as eventually the IDAgeHtWt message will have expressions to handle incomplete knowledge\(^{10}\). The author can check FoundTarget in the Condition of the rule,

\(^{10}\text{Know Value turns out to be seldom used in practice.}\)
but does not. The `Know` predicate cannot succeed unless `FoundTarget` is true, so the check is redundant. Formally, the Condition is present, but in this rule it degenerates to the single conjunct `true`. The action of the rule binds `Target` to `IDAgeHtWt`, making the message available for inclusion in the document. The second rule checks only `FoundTarget` in the condition. If `FoundTarget` is false, then the register’s `missedTargetRule` sets the built-in `_ErrorMessage` string, and the current content rule binds that string to the `ErrorMessage` constituent. Because the success of the first rule implies `FoundTarget` must be true, and the second requires `FoundTarget` false, the rules are exclusive and exhaustive. With only these two rules, exactly one of `IDAgeHtWt` or `ErrorMessage` can and must appear in the final document.

Two further points about mode content rules deserve mention before seeing how the rules are built in MORCONED. First, order of evaluation matters in these rules only in the case that two rules can assert the same non-repeatable message. In that case the success of the first rule evaluated will block the success of the second rule, as the non-repeatable message will be unavailable for reassertion. Second, the rule taxonomy present in FRED is not continued in MORCONED. All content rules in DEXTER are introspective in the sense defined in FRED; they access only the internal scheme, never data in the external scheme. The field fully encapsulates external data access.

**Building a rule in MORCONED.** The author launches MORCONED and makes sure Reg_Tutor is loaded. MORCONED is divided into two tabbed folders, an upper meta folder and a lower working folder. The meta folder has tabs to access internal scheme objects (the list can be expanded to show the aggregation hierarchy), a tab listing defined content messages (with an option to view details of parameters and comments), and a tab where new rules may be created, saved, and so forth. The working folder contains tabs for defining the standard predicate lists, for defining the condition, for binding internal scheme objects to messages, for storing actions of the rule, for viewing messages returned when rules are executed, and for seeing the results of rule execution. The mechanism of specifying a rule, saving it, generating it as a Java class, compiling that class, and executing the rules either individually or in lists is similar, but not identical to, the mechanism in FRED. Most of those actions are taken through the `BUILD` menu.

Aside from small differences in interface, MORCONED has a few significant differences from FRED. MORCONED does not allow definition of runtime arguments. Content-rule
processing occurs after field rule processing. Execution of content rules in MORCONED requires that an instance of the internal scheme exist before content rule execution. The instance MORCONED uses is saved to disk as a serialized object by FRED. EXECUTOR also writes the post-field instance of the internal scheme to disk as a serialized object. Therefore, if EXECUTOR encounters a problem while processing a register it MORCONED allows the author to examine the content rules on the exact instance of the internal scheme.

To create the first content rule described above, the author follows these steps.

1. Move to the Rules tab in the meta folder, Click Define and Load New Rule to show the Define new rule dialog.

2. In the Define new rule dialog, enter the rule name, in this case "FirstIdentification", in the text field and click Accept. The rule name appears in Select Rule choice list, and the working folder is enabled.

3. Move to the Internal Scheme References, tab in the meta folder.

4. Click Show Subreferences to see the aggregation hierarchy of the internal scheme.

5. In the hierarchical List of Internal Scheme References select Target.IFName, Target.INName, Target.IAge, Target.ISex, Target.IHeight and Target.IWeight.

6. Move to the Know tab of the working folder to define the standard predicates.

7. Click the KNOW button, causing the six selections in the List of Internal Scheme References list to be copied to the KNOW list and to be deselected in the List of Internal Scheme References Figure 4.20 shows the result.

8. Click Save on the bottom row of buttons in the same Know tab. (MORCONED will silently ignore portions of a rule specification marked CHANGED when it generates a rule. It only includes specification marked working or on disk.)

9. Move to the Assert tab in the working folder.

10. Highlight Target in the List of Internal Scheme References.

11. Move to the Content Messages tab in the meta folder and highlight IDAgeHtWt.
Figure 4.20: MORCONED defining KNOW for FirstIdentification.

12. Click Load Current Message Selection in the Assert tab. The Parameters List on the left of the tab will show Person of type tutIPof, which may not be null, as the sole parameter of IDAgeHtWt.

13. Select the Person parameter from the Parameter List.

14. Click Selected Parameter gets Current Ref. MORCONED warns the author that type compatibility is not checked, and then copies the internal scheme object Target, selected in step 10, to the list on the right of the Assert tab. The presence of Target in the right-hand list indicates that the argument for parameter Person is object Target (see Figure 4.21.)

15. Click Add Assertion to Working Action.
16. Move to the *Action* tab and click the tab’s *Save* button.

17. Move to the *Rules* tab of the meta folder and click the *Save CHANGED Current Rule*.

**Testing the rule.** At this point, the rule is specified as described above, and the specification is saved to disk. The register author uses the *Generate Content Rule* selection on MORCONED’s *BUILD* menu followed by the *Compile Content Rule* on the same menu. Assuming the information on the *Messages* tab indicates success after each operation, *FirstIdentification* exists as a compiled Java class. To execute the rule, the author uses the *Execute this Content Rule* selection on the *BUILD* menu’s *Execution ...* submenu. The *Message* tab in the working folder contains the same kind of information as the *Message* tab in FRED. The *Results* tab shows the list of asserted constituents, with the output of the *express()* method of each argument of each parameter (see Figure 4.22). The whole
process takes someone who has used MORCONED before and knows what rule to write less than five minutes.

MORCONED has two further execution options. First is *Execute list*, which is similar to FRED’s execution list. The author specifies some order of rule evaluation and executes the whole list. This list must be specified before the content rules can be executed outside of MORCONED. Just as MORCONED relies on a serialized internal scheme to execute its content rules, STRUCTED relies on a serialized list of parameterized content messages to evaluate its structure rules. The serialized list is only created by MORCONED’s *Execute list* ... selection, not by executing a single rule outside the list. The second execution option, available with execution of a single rule or execution of a list of rules, is to show content message availability after each rule is evaluated. These results appear in the message window. Since some content messages are not repeatable, but they might have been asserted by any of several rules in a list, this option is a way of finding which rule asserted, and thereby blocked further assertion of, a particular content message.

Creating and testing the second planned content rule (*NoneSuch*) is similar to creating and testing the *FirstIdentification* rule. Once that task is completed, the content portion of the RegTutor register is completed for the sentence currently under construction.

![Table](image)

**Figure 4.22:** MORCONED shows results after executing *FirstIdentification*. 
4.2.5 Determining Structure: PRESED, CODEX, STRUCTED

PRESED and CODEX

The Reg_Tutor document as planned will have little structure. However, the author must define at least one constituent that is a presentation message. He has specified that the expressions of IDgeHtWt are only available in a paragraph context. Therefore, there must be some context that is a paragraph. A paragraph should be represented as a presentation message. The author can use PRESED to build a constituent that is a presentation message, a context, and repeatable, called \texttt{untitledParagraph}\textsuperscript{11}. PRESED varies from CONED in only two ways: A presentation message may or may not be a designated a context, and presentation-message parameters must be of one of the following DexterOntology types: \texttt{DexterString}, \texttt{DexterInt}, \texttt{DexterFloat}, \texttt{DexterBoolean} or \texttt{DexterDate}. Separation of PRESED and CONED is a legacy of DEXTER's development. I do not go into detail on using PRESED, because using PRESED is very similar to using CONED. Describing the presentation messages is important, however.

The author defines two presentation messages, both contexts. \texttt{UntitledParagraph} is a repeatable context with has no parameters. \texttt{AboutTitle} takes two string parameters, \texttt{fname} and \texttt{lname}, either of which may be \texttt{null}. \texttt{AboutTitle} is not repeatable in Reg_Tutor. The author uses CODEX to give these constituents expressions.

\texttt{UntitledParagraph} has one HTML targeted expression, always available, with the text <P>, an HTML paragraph opening. \texttt{AboutTitle} has two expressions. The first requires both \texttt{fname} and \texttt{lname} to be present and has the following text.

```
\n<!-- aboutTitle case fname lname -->\n
<H1>About {fname} {lname}</H1>\n```

The second expression is flagged in CODEX as the \texttt{default} expression. Default expressions are used only when no other expression is available. This expression is used when \texttt{fname}, \texttt{lname} or both are missing. The text of the second expression follows.

\texttt{untitledParagraph} would be the first candidate for a set of built-in constituents in the next version of DEXTER.

\textsuperscript{11}
Once these constituents are built, the author is ready to begin defining the structure of the portion of the document under development. He uses STRUCTED.

**STRUCTED**

Field rules output an instance of the internal scheme of a register, and content rules output a set of content messages with internal scheme objects bound to parameters. Structure rules evaluate the set of available messages, and the epistemic portion of the internal scheme, to determine which available content messages to include in the document, which presentation messages to add to the document, and the order of all messages appearing in the document. STRUCTED is the STRUCTure EDitor. The output of the structure rules is a list of parameterized constituents interleaved with processing instructions. That list constitutes the document outline. The DEXTER execution engine may then process the outline, selecting and interpreting an available expression for each included constituent.

The form of a DEXTER structure rule is:

**IF ( (Standard Structure-Predicate List) And Epistemic Condition ) THEN Action List**

The *Standard Structure-Predicate List* is a precondition consisting of three kinds of predicates. *Available*(X) is defined where X is any constituent name. *Available*(X) is true if and only if there is a message named X defined and X is available. A message is available if it is a content message asserted during content rule evaluation, or if it is a presentation message defined in the register. *Unavailable*(X) is equivalent to ¬*Available*(X). In the current register development exactly one of *ErrorMessage* and IDAgeHitWt is *Available* in the current Reg_Tutor. One will be asserted and satisfy *Available*, one will remain unasserted and satisfy *Unavailable*. The third kind of predicate is *Open*(X), where X is the name of a context. *Open*(X) is true if X is on the context stack, meaning X has been opened but not closed. There may be only one instance of the *Open* predicate evaluated in an individual structure rule, but arbitrarily many instances *(Un)*Available predicates
may be evaluated. In STRUCTED, rules these predicates are evaluated with reference to message name only, without reference to argument values.

The Epistemic Condition is a conjunction of Boolean (epistemic) objects of the internal scheme. The objects may be negated. A structure rule is predicated on only the epistemic portion of the internal scheme. For instance, a structure rule cannot check that Target.Age > 50. Changing the order of messages based on data values requires either two different messages to express the data in two different value ranges, or it requires an epistemic variable indicating which range contains the value. In the latter case, two structure rules can be written to respond to the knowledge of the data value rather than the data value itself. For example, if we wish to say that a statement about a person having many pets should appear before FirstIdentification in Reg_Tutor, we can set an epistemic object true if the number of pets exceeds our threshold, and false otherwise. The structure rules can then use this epistemic object to decide to place a message about pets before, or after, FirstIdentification.

The Action List of a structure rule supports various actions, including execution of verbatim Java code. Content messages may be reasserted by name, without reference to arguments. This restriction has the effect that a repeatable content message cannot be repeated unless the message is parameter free; a parameterized content message cannot be repeated with varying parameters in the same document. The reason for and the effects of accessing content messages only by name in structure rules are discussed in Section 6.1.2.

The Action List of a structure rule can assert presentation messages. Because structure rules perform the initial assertion of presentation messages, internal scheme objects or literal values may be passed as actual parameters. If a presentation message is a context, then its assertion has the added effect of opening the context. A context is opened by placing its name on a stack. The context remains open so long as its name remains on the stack. Structure rules may explicitly close a context. When a structure rule closes a context, each context is popped from the stack until the context to close is popped. A structure rule may also block a particular constituent by name, meaning that the named constituent will no longer be found available by subsequent structure rules. The message-blocking mechanism allows a rule to indicate that a repeatable message is no longer repeatable or that a particular message has become inappropriate. Structure rules can issue an instruction to set a focus. The immediate effect of an instruction to set a focus is that the requested focus is
unset. An expression cannot set an already set focus, so unsetting the focus insures that the next expression that can set the named focus will do so.

Order of structure rules is important. Even when there is no other relationship between two structure rules, the output of the rule that succeeds first appears first in the document. STRUCTED requires the author to define a tree of structure rules, rather than a list, to control order of rule evaluation. I defer discussing the tree until we have looked at the necessary structure rules for the current train of thought of the Reg_Tutor register.

**Three Structure Rules.** At this stage, the author needs only three simple structure rules for Reg_Tutor. One asserts the **aboutTitle** message, one the **IDAgeHtWt** message, and one the **ErrorMessage**. It is feasible to write two rules, one handling **aboutTitle** and **IDAgeHtWt**, the other **aboutTitle** and **ErrorMessage**. In that case, the tree is simpler, but the rules are more complex. I prefer simpler rules.

The first rule, putTitle, follows.

```plaintext
IF ( Available( aboutTitle ) AND FoundTarget ) THEN
    assert context aboutTitle( fname=Target.FName, lname=Target.LName )
    close context aboutTitle
```

Including **FoundTarget** in the epistemic condition guarantees that **Target.FName** and **Target.LName** will be usable. STRUCTED is smart enough to recognize that a message is available only if its class file\(^\,\,12\) exists. It makes sense to close **aboutTitle**, since it is unlikely there should be more than one message in the title, though multiple-message titles are possible.

The second rule, IdentifySubject, is

```plaintext
IF ( Available( IDAgeHtWt ) AND true ) THEN
    assert context untiltedParagraph
    SET SUBJECT
    assert content IDAgeHtWt
```

---

\(^12\)In this case the file `classes/Dexterpackage/Mode/Reg_Tutor/Constituents/ContentMessages/putTitle.class`. 
UntitledParagraph must be asserted because the expressions associated with IDAgeHtWt require a paragraph context. The subtitleParagraph context should be left open, because in all probability subsequent messages (for the next two sentences) will also require a paragraph context. The SUBJECT is likely to be unset when this rule asserts IDAgeHtWt, so the instruction SET SUBJECT is not strictly required here. It is good form, however, and necessary in general to issue an instruction to set a focus. There is no necessary epistemic condition in this rule. If IDAgeHtWt is available then FirstIdentification succeeded during content rule evaluation, meaning FoundTarget is true. Although there is no harm in checking FoundTarget, the epistemic condition of this rule may be left as the degenerate case (a single conjunct, true).

The third rule is indicateError. It looks like this:

IF ( Available( ErrorMessage ) AND true ) THEN
    assert content ErrorMessage

As the register evolves and more errors become possible, this rule’s condition may become more complex.

Defining a rule in STRUCTED. As usual, the author begins by launching STRUCTED and making sure the current register is Reg_Tutor. The general layout is similar to other editors: meta folder and working folder, supplemented primarily by a BUILD menu. There is also a TENOR menu, which I discuss when describing how to test rules.

The meta folder holds four tabs. The Rules tab is similar to the Rules tab in MORCONED. The Internal Scheme References tab adds only one feature to the same tab in MORCONED. In STRUCTED, there is a switch to restrict the internal scheme objects displayed so that only Boolean values are shown. When building an epistemic condition the Boolean-only view is convenient. The Content Messages tab is similar to the same tab in MORCONED. However, the Available and UnAvailable buttons appear in this tab, while in MORCONED the standard predicate buttons appear in the Know tab in the working folder. Finally, the Presentation Messages tab is new with STRUCTED. It contains separate lists of context and non-context presentation messages. The functionality for each of these lists is the same as for the content message list on the Content Messages tab. There is a button, however, to copy a selected context from the context list to the Open Context field on the Available
tab in the working folder. Placing a context in the Open Context field means that the rule cannot succeed unless the named context is already open.

The working folder has six tabs. The Messages, Results and Action tabs are all as in MORCONED. The Assert tab is also similar, except that there are options to assert content, context or non-context presentation messages. Passing objects to parameters of content messages is disabled on STRUCTED’s Assert tab. In structure rules asserting a content message places the message in the document outline. Content rules determine the actual parameters of content messages. The Epistemic Condition tab allows only Boolean objects from the internal scheme, possibly negated, to be placed in a list. The epistemic condition of the rule is the conjunction of all literals in the list. The Available tab contains the Available and UnAvailable lists, and the Open Context field, all of which allow the author to define the standard structure-predicates described above.

The BUILD menu is similar to the build menu in MORCONED with one exception. The Execute list... option on the Execution ... submenu in MORCONED is replaced by a Rule Tree option appearing directly on STRUCTED’s BUILD menu. I discuss the Rule Tree selection under “Building a rule tree” below.

Once the author launches STRUCTED, the following steps create the IdentifySubject rule.


2. Enter IdentifySubject in the Define Rule text field and click Accept.

3. Open the Content Messages tab and select the IDAgeHtWt message.

4. Click Available.

5. Open the Assert tab in the working folder.

6. Open the Presentation Messages tab in the meta folder and select untitledParagraph from the Context Presentation Messages list.

7. Click Load Selected Context (Pres. Msg) in the Assert tab.

8. Click Add Assertion to Working Action in the same tab.
9. Select the *Action* tab and click the *SET SUBJECT* button. *SET SUBJECT* will appear in the action list. Select it and then click *move down* so that *SET SUBJECT* appears below, rather than above, the assertion of *untitled Paragraph*.

10. Select the *Content Messages* tab in the meta folder and select *IDAgeHtWt*, if it is not already selected.

11. Under *Assert Selected* on the *Action* tab, click *Content Message*. The *Assert* tab comes to the foreground. On that tab click *Add Assertion to Working Action*.

12. Click *Save CHANGED Rule* on the *Rules* tab.

13. Select the *Action* tab. The editor should appear as in Figure 4.23.

Compilation and testing of *IdentifySubject*, using the *Execute this Structure Rule* option on the *Execution* submenu of the BUILD menu, is now possible. The *indicate Error* and *putTitle* rules may be built in the same way. Following compilation of all three rules it is time to build a structure-rule tree to test the rule set.

**Building the rule tree.** The *BUILD* menu's *Rule Tree* selection leads to the *Rule Tree* dialog. The *Rule Tree* dialog provides a graphic approach to building a tree of structure rules. It is generally straightforward to use. Rule trees capture the potential organizations of a document. Each branch of the tree represents one selection and ordering of constituents. *STRUCTED* defines rule trees by name so that the same set of structure rules may support several families of document organization. Trees must be saved and loaded by name. Definition, selection and removal of trees is handled through the *Rule Tree* dialog. For the *Reg Tutor* register, the first steps of defining a rule tree follow.

1. Select the *BUILD* menu's *Rule Tree* option to show the *Rule Tree* dialog.

2. Click *New Tree*, enter the name *Reg Tutor T1* in the dialog and click *Accept*.

3. The first element in the document outline should be the title, so select *putTitle* from the list of three rules and click *Add Selected Rule (below) to Tree (above)*. The state of the dialog at this point is shown in Figure 4.24.
The IdentifySubject and indicateError rules may be added to the tree in the same way. The rule level controls are used to create the tree as it appears in Figure 4.25. Figures 4.25 and 4.26 represent the same tree.

A document outline is created when a depth-first traversal of the tree reaches a leaf node. Traversal begins at the root of the tree. The putTitle node (rule) is tried. If it succeeds, then its leftmost (in this case only) child, identifySubject, is tried. If identifySubject succeeds then a document outline is complete. If putTitle fails then the root’s next left-most child, indicateError is tried.
Figure 4.24: Beginning the rule tree.

Suppose that putTitle succeeds, but identifySubject fails. In this case the author might want the document to be considered successfully terminated, if abbreviated. He may highlight putTitle in the tree and click Rule has Success Child. Formally, all branches that form a document outline end in a special node, silent in the document, called success. Suppose that the indicateError node (rule) is tried and somehow fails. Any node can be given either a success node right-most child or an error node right-most child (never both). A branch ending in the error child is a failed document. The tree now described is indicated in Figures 4.27 and 4.28, which are equivalent. PutTitle could also be given its own error child, rather than a success child.

The Ignore When Rule Fails button may be combined with either Rule Has Success Child or Rule has Error Child. “Ignoring” is a form of cut, as in PROLOG [11]. In the context of structure rule trees, ignoring rule failure captures the notion that, after a certain prefix of a branch has been successfully traversed, backtracking should no longer occur. In essence, the constituents on the rest of the branch are optional. The author can ignore rule failure to avoid constructing two branches, one without the post-cut branch suffix and one with
The rule tree serves the same functions as the rule lists in FRED and MORCONED. The tree allows the author to test the rules in the correct order while working in STRUCTED. It also defines the order of rule execution when the register is used to process data outside the editing environment, and constructs a document outline as a serialized object on disk.

**Testing the rules.** Using the rule tree to test rules in STRUCTED requires the author to create the tree as described above and to use the *Rule Tree* dialog *Save* and *Execute Tree* buttons. Results appear in two forms: in the working folder’s *Messages* tab, showing
messages and possibly constituent availability as before, and in the same folder’s Results tab, showing a representation of the generated document outline. Sample results are shown in Figures 4.29 and 4.30 respectively.

At this point, the author has completed the first portion of his document. By using STRUCTED’s TENOR menu, he can open the Tenor Window and see the generated document (in raw HTML in this case) by clicking Process (Figure 4.31).

The author can now work on the second and third sentences of the document, the inclusion of a photo when available, and the expansion to data unlike that associated with Victor March. I do not show each step of the process, but do briefly revisit the register’s field, before exhibiting the completed Reg_Tutor register.
4.2.6 Revisiting FRED for an Iterative rule and a Take-first rule

Suppose the author now wishes to provide field support for his second sentence, represented by, “He and his wife, Elizabeth, own a cat named Carmen.” Because I recommend heavy use of epistemic variables in DEXTER, I suggest six more epistemic variables: 

SPouse.DEFINED, IS.MARRIED, SHARE.INAME, PETS.DEFINED, MULT.PETS, HAVE.PHOTO. They make subsequent rules easy to read, and STRUCTED requires Booleans to examine the internal scheme. NUM.PETS (an integer) and PhotoLocation (string) are also needed. In addition to these objects, the author defines eight more field rules. (These eight, however, cover the representing the spouse, pets, pet preferences for both the person of focus and the spouse, and photo of the person of focus, for all individuals, not just those like Victor March.) The rules can be summarized as follows:

FindMaritalStatus: singleton,
    defined on getPerson,
Figure 4.31: The HTML document displayed in the Tenor Window.

assigns IS_MARRIED and Target.IMaritStat

FindSpouse: singleton
    defined on getSpouse,
    assigns Target.ISFName, Target.ISLName, SPouse_DEFINED and SHARE_LNAME

assignSpouseEmpty: introspective,
    assigns Target.ISFName, Target.IILName empty

FindPets: iterative,
    defined on getPets,
    assigns NUM_PETS, PETS_DEFINED, MUTL_PETS, Target.IAnimalsOwned

assignPetsEmpty: introspective,
    assigns Target.IAnimalsOwned empty

assignTarget: introspective,
    a catch-all rule to be sure Target is marked assigned if
    Target’s ISFName and IILName members are known
FindOnePhoto: take-first
    defined on getPhoto,
    assigns PhotoLocation and HAVE_PHOTO

assignPhotoEmpty: introspective
    assigns PhotoLocation empty

Iterative rules and take-first rules, described in Sections 3.5.2 and 4.2.3, have not appeared previously in Reg_Tutor. The examples of iterative and take-first in Reg_Tutor rules merit some comment.

FindPets is a DEXTER iterative rule. Its precondition and condition appear as in Example 4.13.

**Example 4.13** FindPets (precondition and condition)

```
IF ( [ getPets:Kind IS_NOT_NULL AND getPets:Name IS_NOT_NULL ]
    AND FoundTarget AND Target.IAnimalsOwned IS CREATED )
```

The condition includes Target.IAnimalsOwned IS CREATED. The member IAnimalsOwned is a DexterArray and must be specially initialized. The initialization SET IAnimalsOwned = 0 in the initialization Target suffices. Initialization forces the array object to be created when the Internal Scheme is initialized. Objects may be initialized using the FRED OBJECTS menu. The creation condition should be checked because if it is not met a rule using the array fails at runtime. This situation is an inconvenient legacy of DEXTER's early development.

The assignment of an iterative rule may have three kinds of actions. The assignment portion (or action list) of the FindPets rules is shown in Figure 4.32.

Iterative rules capture the notion of programming with loops. The ultimate goal of an iterative rule is to evaluate each tuple of a result. Each time the rule succeeds on a tuple, some actions in the rule's action list are executed. In Figure 4.32, those actions are the ones in the middle block, setting values for Target.IAnimalsOwned[ _currentRow ].IKind and Target.IAnimalsOwned[ _currentRow ].IAnimalName. FRED provides three built-in objects to help with iterative processing: _currentRow (the zero-based index of the tuple being evaluated, here used as an array index), _successCount (a count of the tuples for
which the rule succeeds) and _rowCount (a count of the total number of tuples in the result). Some actions need to be taken before processing any tuple, rather than at each successful evaluation of a tuple. For example, before it fills an array, an action must allocate the necessary elements. That task is performed by the line PRIOR SET Target.IAnimalsOwned._rowCount. The PRIOR directive captures the notion of executing an action before the loop processing begins. The complement of prior is TERMINAL, which indicates an action should be taken only after the rule has tried all tuples. Since the value in _successCount vanishes after an iterative rule executes, FindPets stores the result in NUM_PETS.

FindOnePhoto is a take-first rule. If the database has zero or one photo of each person, a singleton rule correctly finds and stores the location of any available photo. However, if more than one photo is possible, then a singleton rule has the wrong semantics; it fails on more than one tuple. If, as in the Reg_Tutor case, the author only wants one of the possible tuples, he can define a rule just as if it were a singleton rule, but he uses FRED's New Rule dialog to indicate that it is a take-first rule. Take-first semantics accept more than one tuple in a result but only process tuples until the rule's condition is satisfied one time. At that point the action list is executed and no further tuples are evaluated.
4.3 Reg_Tutor Completed

Reg_Tutor outputs for the five cases described in Section 4.2.1 are displayed in Figures 4.33 through 4.37. One additional case in is shown in Figure 4.38.

About Victor March

Victor March is a slightly overweight old fellow who is of average height. He and his wife, Elizabeth, have one pet named Carmen (see). He likes cats; Elizabeth likes dogs.

Figure 4.33: About Victor March, shown in the completed register.

About Joseph Oldman

Joseph Oldman is an overweight middle-aged fellow who is of average height. Mr. Oldman is married to Dianna Hobart. They have no pets. He likes fish; Dianna’s preferences are unknown.

Figure 4.34: About Joseph Oldman, shown in the completed register.

About Miroslaw Szczygielski

Miroslaw Szczygielski is a middle-aged man who is tall and slightly overweight. He and his wife, Helena Szczygielska, have no pets. His preferences are unknown. Helena’s preferences are unknown.

Figure 4.35: About Miroslaw Szczygielski, shown in the completed register.

The author should address several questions. Do the first five results give a reasonable approximation to the documents in the original plan? What are the reasons for any differences? What does the final document show?
About Beth Silverstein

Beth Silverstein is a thin middle-aged gal who is of average height. Ms. Silverstein is married to Raphael Wrinkle. They have only one pet, Stalwart (bogey). The boys prefer badges; Raphael likes badges.

Figure 4.36: About Beth Silverstein, shown in the completed register.

About Natalia Sturek

Natalia Sturek is a young woman who is of average height and slim. Ms. Sturek is not married. She no pets. Her preferences are unknown.

Figure 4.37: About Natalia Sturek, shown in the completed register. Notice “She no pets.”

Whether the first five results are reasonably similar to the original documents is ultimately a subjective judgment. They are certainly in the right spirit. In the case of “Victor March” (Figure 4.33) the expression uses “fellow”, rather than “male.” The expression could have chosen “male.” Registers can introduce this kind of variability when the author desires. However, the register as written cannot say “a cat named Carmen” nor can it say “... but Elizabeth prefers...” The reason in both cases is inadequate array handling. CODEX expressions cannot access individual array elements. This deficiency can be corrected in DEXTER in one of two ways. CODEX expressions could be given direct access to array elements, including the ability to iterate over an array. Alternatively, an implemented class editor could define express() methods using CODEX-style expressions, and endow the arrays themselves with better expression. As it stands, “Carmen (cat)” is stilted, but grammatically (barely) acceptable. The register can be modified in DEXTER to treat
arrays of pre-determined length as special cases. At least for the special case of a single preference, it would be possible in the field to determine (introspectively) if the preferences were the same. With that (epistemic) information the phrase, "... but Elizabeth likes ...", could be generated. The same method would handle "Both Beth and Raphael like badgers.", which is a goal for the "Beth Silverstein" document. In the currently generated document, "She likes badgers; Raphael likes badgers" is awkward (Figure 4.36.) The "Joseph Oldman" and "Miroslav Szczypielski" documents (Figures 4.34 and 4.35) are close to the original documents. In "Miroslav Szczypielski" the failure to aggregate the unknown preferences of the couple is analogous to the failure to aggregate the identical preferences in "Beth Silverstein." The variance of "Natalia Sturek" (Figure 4.37) with the original document is incidental, and can be corrected if the author deems it a problem. The text in Figure 4.37 includes "She no pets." Such errors can occur in canned text interleaved with computed values. At present CODEX does not help the author find such errors. The execution engine does, however, allow CODEX expressions to be revised without re-compilation of the register. "About Beth Silverstein", similarly shows an omitted period. The final displayed document, Figure 4.38, "Brenda Chan" again illustrates that array expression is weak.

The final register uses 36 field rules, 19 objects, 5 content messages, 2 presentation messages, 5 content rules and 6 structure rules. The 7 constituents have a total of 12 expressions, though all of the content messages would do well with more. The one defined rule tree appears as in Figure 4.39, representing an almost linear structure. The document is deemed successful if it says anything at all. To bring the register to this point would take an experienced author about 8 hours, of which approximately 4 hours would be required to build the field, and about 3 hours to build the expressions. Experience shows that altering output is relatively easy. The author can alter expressions on-the-fly.
Figure 4.39: Final rule tree for Reg_Tutor.

**Stand-alone Execution with EXECUTOR.** EXECUTOR is a means to execute a register outside the editing environment. For the Reg_Tutor register, either of the following invocations suffices, assuming “Victor March” is of interest:

```make
make REGISTER="'Reg_Tutor'" ARG="'Victor' 'March'"
```

or

```java
java Executor Reg_Tutor Victor March
```

When a DEXTER register defines multiple rule trees, the user may specify the name of a tree on the command line. See Chapter 5 for further examples.
Chapter 5

More Registers

I provide examples of four registers in addition to the Reg_Tutor register discussed in Chapter 4. These registers deal with a midterm-grade report, facts about universities, and the status of a computer network (two registers). Taken together, these examples show that registers are applicable across a range of domains. Because DEXTER was used to build each register, these registers also show that a single register system can be used to author diverse registers. For registers, the form of the data and the complexity of user expectations are more important than superficial semantic similarity of domains. In all of these examples the data fit into a flat relational scheme (miniSQL tables) and the reports are fundamentally simple. The requirement of a flat relational scheme is specific to the first version of DEXTER, and need not hold for other register systems. Domain differences are more important in traditional NLG approaches [32, 68]. Whether registers scale up beyond simple documents is an open question not addressed by my research.

5.1 Midterm Student Grade

The Midterm Student Grade register develops a midterm grade report addressed to a student. There are eight tables pertinent to any generated report. Grade_Scale is keyed by a class identifier and contains cut-off points for grades of A, B, C, and D. The table Personal carries personal student information including gender. Teaching associates instructor names with courses. ImportantDates simulates a registrar's online record of significant dates through a semester. The Assignments table is again keyed by a class identifier and records assignments by type (for instance homework or quiz), index (first
homework, second homework), a maximum possible score and whether that the assign-
ment is graded yet or not. The Weights table records weights for each assignment in each
course. For each class there is a table, for example CS.999.000, keyed by a student iden-
tifier. It records the class identifier, assignment identification and recorded score for each
assignment a student completes in a class,

The Midterm Student Grade register consists of 36 field rules, 24 internal-scheme objects,
8 content messages, 9 content rules, 6 presentation messages and 12 structure rules. The
register uses two rule trees, shown in Figures 5.1 and 5.2. Corresponding sample outputs
of the two trees are shown in Figures 5.3 and 5.4. The first invocation is

```
make REGISTER="'Mid Term Student Grade'" VERSION=-vGoal_Tree
ARGS="'CS 999 000' 'Minnie Mouse' CS_999_000"
```

The second invocation substitutes Simple_MTG_Report for Goal_Tree.

```
Figure 5.1: The Goal_Tree rule tree of the Midterm Student Grade register.
```

```
Figure 5.2: The Simple_MTG_Report rule tree of the Midterm Student Grade register.
```

Registers allow highlighting significant points, for example in the Midterm Student Grade
register the actual grade, if the final representation language provides some means of doing
so. Highlighting can be conditional on data value. For example, the author could choose
to highlight either an A or an F, but no other grade. Registers also allow statements to
be reorganized based on data value. For example, in the Midterm Student Grade register,
if a student is found to be missing too many assignments, say more than 2, then the
Report For CS 999 000

Dear Ms. Mouse

The instructor's records show you are enrolled in CS 999 000. This is your midterm grade report. You are advised to review this report and address any issues as soon as possible.

Your grade in CS 999 000 at this point is B. It could be considered a high B.

Letter grades depend on weighted numeric scores. Your current weighted score, calculated based on all scores recorded, is 87.8. The following marks are recorded in your name:

Hw1 71, Hw2 79, Hw3 69, Hw4 189, Qz2 50, Qz1 50, Mx1 90.

You have 0 missing assignments. Please review this report and address any concerns before the last day to withdraw.

Figure 5.3: Output using Goal_Tree in Figure 5.1.

Midterm Grade Report for CS 999 000

Dear Ms. Mouse,

This is your midterm report for CS 999 000. These are your recorded marks:

Hw1 71, Hw2 79, Hw3 69, Hw4 189, Qz2 50, Qz1 50, Mx1 90.

At this time your assigned grade is B.

Conclusion

Having reviewed this report, please address any concerns before the last day to withdraw.

Figure 5.4: Output using Simple_MTG_Report in Figure 5.2.

remark about missing assignments can be moved higher in the document to give it more prominence, as in Figure 5.5.

In Figure 5.5, the last day to withdraw appears twice. In Figure 5.1, built with the same document tree, the last day to withdraw does not appear. If the register author finds the inconsistency or redundancy unsatisfactory he can correct it by adjusting expressions.

The Midterm Student Grade register shows that it would be useful to allow a register to query more than one information source. In this register, the miniSQL database simulates a registrar’s online information in its Personal, Teaching and ImportantDates tables, and an instructor’s online information in its CS_999_000 and like tables, as well as in its Grade_Scale, Weights and Assignments tables.
5.2 University

The University register was designed to see if registers can serve as output managers for systems focused on gathering data, rather than presenting information. Angel Janevski's tool, UniversityIE, is a semantically enhanced web crawler [34]. One search target, from which UniversityIE draws its name, is universities; it derives such information as the address, and tuition.

The University register has 9 field rules, 10 internal-scheme objects, 18 content rules, 10 content messages, 13 structure rules, 3 presentation messages and one rule tree. It is based on three tables, CAMPUS, TUITION and REQUIREMENTS. Sample outputs are shown in Figures 5.6 and 5.7. Both invocations (this time not using \texttt{make}) were as follows: \texttt{java Executor University 'University of Kentucky'}. The data does not change between register executions, but the surface appearance of the documents does change. In Figure 5.7, tuition appears in a table, while in Figure 5.6 it does not. The only difference is that different expressions were chosen. The messages (hence the document semantics) are the same. Allowing this degree of variability in expressions is probably a bad idea for most applications, but it is feasible. At the time the University register was built, tabularization depended on an expression to explicitly include the HTML table-creation commands. A register system that supports a document-representation language like HTML should include a simple table command. DEXTER has since added explicit, declarative support for HTML tables.

The University register demonstrates two particular points. First, registers are condition-
About The University of Kentucky

This university is contained on one campus. The campus is located in Lexington KY.

Cost And Aid Data

This University offers financial aid. For in-state students, tuition rates are: Full Time 1460; Part Time 153. For out of state students, tuition is as follows: Full Time 4180; Part Time 446.

Requirements / Application Deadline

This University requires a GPA of 2.5 and a TOEFL (Test Of English as a Foreign Language) score of 550 for foreign students. The application deadline is 3/1/99.

Figure 5.6: University register output, same data as Figure 5.7.

<table>
<thead>
<tr>
<th>Tuition Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>HS State</td>
</tr>
<tr>
<td>Full Time</td>
</tr>
<tr>
<td>Part Time</td>
</tr>
</tbody>
</table>

Figure 5.7: University register output, same data as Figure 5.6.

ally monotone. UniversityIE can alter its search to add new information to the database, even expand the database scheme, but the register does not change. The register remains unaffected so long as the same queries contain the same form of result. This form of monotonicity can fail if, for instance, a register assumes a particular format for a string representing an address, and that format changes. The query would remain valid, the result format unchanged, but the rules' handling of the result would be in error. A register should not usually rely on the format of a string or the like. External data should usually be treated as atomic. Failing to treat external data as atomic risks a fragile register.

The second point illustrated by the University register is that, when the system providing the input data does not provide a query mechanism, an intermediate piece of software must provide queries. For instance, if a search engine does not report its results in a miniSQL database, then DEXTER registers cannot access the results directly. Something must translate the results into a set of miniSQL tables. In the University case, initial search
results have been simulated, although UniversityIE should be able to write miniSQL tables soon.

5.3 Pulsar

Pulsar is a software tool for keeping a Unix network administrator abreast of what is happening on the network [19]. Pulsar has two fundamental ways of showing results. Anyone who has access to an X terminal can view a graphic display of Pulsar's data as it comes in. Alternatively, a shell script (query.sh) can be run to accumulate the results of recent reports as plain text. An example of the plain text output of query.sh is shown in Section 1.1. The output of query.sh serves as input to two related DEXTER registers, Pulsar_MOTD and Pulsar_Adm. Pulsar_MOTD (Pulsar Message of the Day) parses down the sometimes voluminous output of query.sh to just that information of interest to a network user and then presents that data in a pleasant way as an HTML document. Examples of what might be of interest to a user include the meaning of names (“thelma” is a machine, “diane” a printer), down machines, disks near capacity, and busy machines. Pulsar_Adm (Pulsar Administrator) expands on Pulsar_MOTD by including certain key features such as security alerts, strange processes, missing processes, as well as many (not all) of the features of interest to users. For example, network administrators care about failing disks, but can distinguish printers from machines without any help.

The first problem the author faces in building any Pulsar register is that DEXTER only knows about miniSQL databases, and Pulsar does not communicate its results to a miniSQL database. I solved this problem by writing a small program (pulsar2mysql) to parse query.sh output into miniSQL tables. Field rules can run verbatim Java code, but I extended the possible actions of a field rule to executing an arbitrary program. The first field rule in the Pulsar_Adm register is an introspective rule with a condition of true and one action: executing pulsar2mysql.

An example of the output of the two registers is shown in Figures 5.8 (MOTD) and 5.9 (Adm). The outputs shown in these Figures are the outputs of the two registers on the same data. In this case the invocations were make PULSA and make PULSM respectively. The Pulsar_Adm register shows some evolution in DEXTER. When Pulsar_MOTD was written, DEXTER did not support building tables from arrays in expressions. By Pulsar_Adm, DEXTER had that ability. In these registers there can be lengthy lists of rela-
Pulsar Message Of The Day

This document reports the latest information we have on machines and printers in our system. When problems occur such as resources being unavailable or using backup power, disks nearing capacity or machines having busy cpus, we will report that information. These 19 names: czarny, diane, grug, indigo, jelah, marel, martha, math, mirek, multilab, neon, mani, raphael, henrik, marvin, graphite, geocad, calvert, speedy, refer to printers. The default interpretation for any resource name in this document is that it refers to a machine name.

Potential Access Problems

Here are the 4 resources that were unavailable (e.g. a down machine) at last report: bloodroot, hunting, galahad, peony.

Disk Concerns

A total of 7 disks were reported to have a capacity problem: on damas: disk / at 97% of capacity; 126440 left, on earth: disk /usr at 98% of capacity; 5347 left, on golf: disk /home at 100% of capacity; 7568 left, on heinlein: disk /usr at 98% of capacity; 16026 left, on herkimer: disk /usr at 98% of capacity; 14396 left, on norton: disk /usr at 98% of capacity; 3323 left, on state: disk /usr at 96% of capacity; 7733 left. Disk capacity left is reported in KB.

Busy Machines

Two reports indicating a busy machine: on herkimer: load is 1.00 ; 0.2% total time 00:04:23 root /etc/X11X pid 530 ; 98.8% total time 05:16:52 raphael /local.exe pid 762 (on Fri at 3 pm), on kocyn: load is 1.00 ; 99.6% total time 11658:45 klopper 2cac pid 10656 (on Fri at 3 pm). (Busy machines are noticed by having a high load average).

Figure 5.8: Pulsar_MOTD output on the same data as Figure 5.9.

tively long text strings describing disk capacity problems and so on. In Pulsar_MOTD, as shown in Figure 5.8 under Disk Concerns, these lists are expressed as comma-separated lists, even though a comma may appear in any list element. In Pulsar_Adm, any list of more than two disk problems is put into a table (Figure 5.9). Pulsar_Adm does not show great variability in expression. It generally uses the text found in the database directly, since Pulsar’s pulse monitors are extensible and output string formats could easily change. In Pulsar_Adm and Pulsar_MOTD the variability is using a table or not, and sometimes excluding a section if, for example, there were no down machines, or were no software problems such as missing processes.

The most interesting aspect of the Pulsar registers re-use of resources. Pulsar_MOTD, being the simpler register, was built first. Derivation of Pulsar_Adm followed. However, due to a design flaw in DEXTER’s approach to generated Java classes (Section 6.1.2), re-using, or inheriting, rules, internal-scheme objects, queries or ontologic classes between registers is implemented by copying. Copying must be done by hand, and the classes adjusted just enough to make them accessible in the new register. I did not change functionality of any class and then treat it as inherited or re-used. Nonetheless, these registers give evidence
that significant re-use of resources among related registers is possible. Overall, counting field, content and structure rules, constituents (though not expressions) and internal scheme objects, 67% of Pulsar.MOTD resources were used in Pulsar.Adm, accounting for 44% of the resources used by Pulsar.Adm. Breakdown of resource re-use from Pulsar.MOTD to Pulsar.Adm is shown in Table 5.1.

<table>
<thead>
<tr>
<th>resource</th>
<th>MOTD</th>
<th>Adm</th>
<th>reused</th>
<th>MOTD re-used</th>
<th>Adm supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal scheme objects</td>
<td>17</td>
<td>30</td>
<td>12</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>field rules</td>
<td>7</td>
<td>11</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>overall field</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>content messages</td>
<td>8</td>
<td>12</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>content rules</td>
<td>7</td>
<td>11</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>overall content</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>presentation messages</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>structure rule</td>
<td>10</td>
<td>16</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>overall structure</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>62%</td>
<td>35%</td>
</tr>
<tr>
<td>all components</td>
<td>-</td>
<td>-</td>
<td>35</td>
<td>67%</td>
<td>44%</td>
</tr>
</tbody>
</table>

Table 5.1: Re-use of resources in deriving Pulsar.Adm from Pulsar.MOTD.

Pulsar.MOTD is based on 6 queries and was built in approximately 10.25 hours. The figure of 10.25 hours includes only time working with the implemented portions of DEXTER\(^1\). Pulsar.Adm re-uses 4 queries from Pulsar.MOTD and adds 6 more. Pulsar.Adm required 5.2 hours work directly with DEXTER. Table 5.2 shows a breakdown of times involved with each register. “Other” includes time to simulate query and class editors, time to copy Pulsar.MOTD classes by hand and make them accessible in Pulsar.Adm, and some work on DEXTER tools. The Pulsar.MOTD ontology contains two classes. Pulsar.Adm uses the same classes without modification.

<table>
<thead>
<tr>
<th>Time</th>
<th>MOTD</th>
<th>Adm</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEXTER</td>
<td>10.25</td>
<td>5.2</td>
</tr>
<tr>
<td>other</td>
<td>13.0</td>
<td>5.3</td>
</tr>
<tr>
<td>total</td>
<td>23.25</td>
<td>10.5</td>
</tr>
</tbody>
</table>

Table 5.2: Time in hours developing Pulsar.MOTD, deriving Pulsar.Adm.

Experience with two related registers and an incomplete register system is insufficient to warrant a claim of victory on the code re-use front. However, it is encouraging that the

\(^1\)Time simulating the ontology and query editors is not counted in the 10.25 hours.
development time for the more complex Pulsar_Adm register decreased by half and that a large percentage of the field resources of Pulsar_MOTD were re-usable, even discounting a complete re-use of the ontology. The field is the most technical portion of a register and takes the most time to build. In this case the field re-used the highest overall percentage of Pulsar_MOTD resources.
Pulsar Admin Report

This document summarizes several Pulsar reports along the following lines:

1. Security alerts: attempts to connect with failed negative result, reports of expected processes found (if any), instances of access control being enabled (if any).
2. Variable hardware failures: media failures (if any), disk loss capacity (if any), reports of power supply problems (if any).
3. Processes expected but not found: accounts as a special case (if it happens), other processes (if any).
4. Currently down machines (those that do not respond to ping, if any).

Security Concerns

Failure to Connect

There were 4 attempts to connect which failed:

<table>
<thead>
<tr>
<th>Machine</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>masterred</td>
<td>Jun 7 17:54:18 masterred in ./plg [3049]: refused connect from 216.70.100.100</td>
</tr>
<tr>
<td>Gateport</td>
<td>Jun 12 09:03:42 Gateport in tine [3090]: refused connect from local, cseng.uga.edu</td>
</tr>
<tr>
<td>etcred</td>
<td>Jun 12 16:14:45 etcred in tine [3138]: refused connect from mntn.cseng.uga.edu</td>
</tr>
<tr>
<td>gateway</td>
<td>Jun 12 16:26:21 gateway in tine [3159]: refused connect from laby.cseng.uga.edu</td>
</tr>
<tr>
<td>mainuno</td>
<td>Jun 12 17:22:49 mainuno in tine [3129]: refused connect from acseng.uga.edu</td>
</tr>
<tr>
<td>all</td>
<td>Jun 14 02:11:17 in ./plg [13294]: refused connect from 4-8:155.102</td>
</tr>
</tbody>
</table>

Hardware Problems

Disk Missing Capacity

A total of 7 disk-capacity problems were reported:

<table>
<thead>
<tr>
<th>Machine</th>
<th>Disk at % Capacity KB left</th>
</tr>
</thead>
<tbody>
<tr>
<td>dennis</td>
<td>disk size 97% of capacity, 126440 left</td>
</tr>
<tr>
<td>earth</td>
<td>disk size 95% of capacity, 2394 left</td>
</tr>
<tr>
<td>gulf</td>
<td>disk at 100% of capacity, 1500 left</td>
</tr>
<tr>
<td>healdle</td>
<td>disk size 98% of capacity, 160378 left</td>
</tr>
<tr>
<td>larkin</td>
<td>disk at 98% of capacity, 143064 left</td>
</tr>
<tr>
<td>nontes</td>
<td>disk at 98% of capacity, 53219 left</td>
</tr>
<tr>
<td>rome</td>
<td>disk at 96% of capacity, 77533 left</td>
</tr>
</tbody>
</table>

Software Concerns

Missing Processors

Mondo is not running on the following 7 machines: dennis, earth, gulf, healdle, larkin, nontes, rome. These 7 machines were not included in the list of running machines. There were no other processes reported missing.

Down Resources

There are 4 resources that were unavailable on this report: hardware, hosting, gateway, proxy. They are most likely down machines.

Figure 5.9: Pulsar Adm output on the same data as Figure 5.8.
Chapter 6

Evaluation, Conclusions and Future Work

6.1 Evaluation

Evaluating an approach to generating documents is difficult. No accepted metric for document quality exists. Generation tasks, from planning content to choosing words, lack a solid theoretical foundation [32]. I cannot formally prove when, or measure how well, computational registers work. Evaluation of registers as an architecture must be qualitative, and to some degree comparative. My architecture, like any other method for generating documents, can at present only be evaluated by experimenting with it. I cannot evaluate registers without looking at DEXTER.

6.1.1 Evaluating DEXTER and Computational Registers

Several distinct registers have been built using DEXTER. In each case, the generated documents meet expectations. Experience with DEXTER demonstrates it is possible to build registers and generate documents in a variety of domains by using a single register system. The registers themselves, shown in Chapter 5, testify that computational registers can produce documents that respond to data, target an audience, and allow for variable and generally correct and acceptable expressions of data. However, DEXTER as described in Chapters 4 and 5 is not prefect, nor are registers as described in Chapters 2 and 3, perfect. I look at problems and solutions throughout the current section.
6.1.2 Implementation issues

DEXTER suffers from a number of different kinds of problems that have little to with the architecture described in Chapter 2. These problems stem from either the implementation of DEXTER or from design decisions described in Chapter 3.

Legacy problems

All DEXTER tools exhibit problems that stem from a long, organic development of a large software system. The tools do not have a uniform interface. Even within tools, the same operation may require different syntax in different contexts. For example, in FRED, to set the Boolean object \texttt{IKnow} true, the line \texttt{SET IKnow TRUE} is required and is case sensitive. However, to initialize that object, the syntax is \texttt{set = true}, again case sensitive. The idea of having to write a separate field rule with condition \texttt{true} which must be evaluated before a register can invoke a preprocessing program (as described in Section 5.3) is awkward. FRED should be extended to allow definition of register pre-processing requirements. Beyond FRED, some tools should be combined. CONED, PRESED and CODEX could be readily and effectively combined, as could MORCONED and STRUCTED\footnote{Other combinations are possible too.}. Some tools are semantically inconsistent: CODEX acknowledges both \texttt{SUBJECT} and \texttt{OBJECT} as foci, while STRUCTED acknowledges only \texttt{SUBJECT}. CODEX should allow expressions to access arbitrary array elements when a parameter is an array and to use regular expressions to form conditions on string values in arguments.

The number of legacy problems could be greatly reduced with a re-implementation of DEXTER. A decrease in authoring time should follow.

Ontology Problems

The DexterOntology classes are adequate in general, although some classes might be added. For example, an associative array class would allow lists of string pairs to be efficiently managed. The approach to arrays in the DexterArray class is awkward and needs to be re-designed.
Working with Rule Sets

All of the rule-authoring tools provide good support for an author writing a single rule. All, but especially FRED, could be much more helpful in managing sets. For instance, in FRED the author can delete an object from the internal scheme. It would not be hard to implement a check for all of the register’s rules that use the deleted object, but at present only the currently loaded rule is checked. Similarly simple algorithms exist for checking if all objects in the internal scheme are assigned by some field rule in a list of field rules, for checking if all messages can be selected by some content rule in a list of content rules, and so on. The inverse information, which rules assign a value to an object or select a message, would also benefit the author in certain situations. Similar algorithms could be implemented to see which possible combinations of absent arguments would result in a constituent lacking an expression. Our limited experience with DEXTER shows the value of all of these algorithms. The results would be more reliable registers and easier authoring.

Completeness

The most serious problem is that DEXTER lacks two key components: an class editor and a query editor. Table 5.2 shows that about 50 percent of the building time for either Pulsar register was spent working outside DEXTER. At least half of that time (25% of the total time\(^2\)) was spent building the classes that should have been created by means of these two missing editors.

A query editor would allow the register author to specify, experiment with, and alter the queries on which a register is founded, all while working within DEXTER. It would be the logical tool in which a register would be initially created and in which runtime arguments to the register would be defined. Coupled with a class editor, a query editor would make ADMIN obsolete.

A class editor would allow the author to specify the members of a class and their types. Assuming the DexterObjectInterface or a similar approach, it is possible to infer answers to questions in the object-attribute dialogs described in Section 4.2.1. The editor could use CODEX to define the behavior of the `express()` method of a class. An object would be analogous to a message, members taking the place of parameters, and the object enjoying

\(^2\)Probably a conservative estimate.
the ability to express itself conditionally. Since expressions can be defined offline, object expression would gain the same advantage.

Vocabulary management is also missing. Experience shows that good vocabulary management could provide benefits beyond user trust. When the author wants to use a literal string value, he must take into account word-form and cannot subsequently choose synonyms. Vocabulary management should make string values easier to use by automating both word form selection and synonym selection, as well as allowing the author to add new entries and new word senses to existing entries. Using a vocabulary tool to help with word-form and synonym selection assumes sophisticated vocabulary manager that would significantly expand the complexity of building, and perhaps understanding, the authoring component of a register system. Re-use of an existing tool such as wordnet [18, 50] would be preferable and perhaps necessary.

Implementing data carrying would be straightforward, but in a pre-production system still seems to provide little benefit. Semantic tagging as in Section 2.4.1 is an area that merits further research.

**Presentation-information problems**

I use the ontology of a register as a means of both re-representing the data and as a means of storing presentation information such as grammatical number and gender. Thus far I see nothing to make me doubt the general approach. However, representing all presentation information through a single interface (the DexterObjectInterface described in Sections 3.9 and 4.2.1) has not worked well. Even with only three attributes, the number of questions that must be answered, either implicitly or explicitly is too large to be convenient. Further, one set of attributes does not fit all entities. For example, if a class represents a physical entity, objects of the class are always count. If the class instead represents an action, the question of countability does not make sense. Trying to decide what value to assign to attributes of an object that does not really possess those attributes takes a good deal of time for a novice (register) author. A family of attribute sets, for example for animate and inanimate entities, for classes that model people-authoring tool versus classes that model other animals, might be more efficient. The family of attribute sets and the attribute sets themselves should probably be extensible on a per-register basis, since quests for fundamental descriptive sets have generally not succeeded.
Repeating messages

I designed constituents to be repeatable, recognizing that instances of a single messages can be distinguished by receiving different arguments. The design of structure rules, however, assumes messages are unique by name. The unique-names assumption is a simplification for implementation. It would be best to re-implement structure rules so that conditions can distinguish instances of the same message based on the values of message arguments.

The inability of structure rules to distinguish messages with different parameter values causes problems. What I call “team registers” are currently difficult to write. For example, it would not be difficult now to write a register to describe the contributions of a particular player in a basketball game, given some information about the team and data roughly equivalent to a box score. It is very difficult, however, to describe the entire team’s performance. Suppose I want to say Player A scored 27, player B 25, and three others 15 each. The difficulty is that the pointsScored( player player_id ) message cannot be distinguished by the actual value of player_id. Workarounds are possible, but awkward. It would be far easier if a structure rule could distinguish between instances of pointsScored based on player_id.

Register derivation

As mentioned in Section 5.3, when I derived Pulsar_Adm from Pulsar_MOTD, I discovered a design flaw in the classes DEXTER generates. The problem is that each rule is a class, each ontology class a class, and so on. Each class belongs to a package named by its defining register. The classes can still be imported (in the Java sense) into other registers, but that is not enough to make them available. The DEXTER runtime environment is compiled before a particular register’s rule or ontologic classes. The runtime environment therefore uses the java.lang.reflect API to turn strings into runtime objects of arbitrary classes. The runtime environment assumes that the current register owns the class, so it gets the package name wrong. It should be possible to alter the runtime environment, or to give it information about the proper register name to use for each class. A better solution is probably to not tie the classes to a specific register by package name.
Context

Contexts work well enough in DEXTER but are over-designed. Experience suggests it is adequate to have kinds of contexts, rather than allow each message name to represent a specific context. In other words, all messages with “title” in the message name would represent a title context. “Paragraph” and others could be treated similarly.

Global awareness

Early on I discovered that individual message expressions do a very good job determining whether to use a masculine, feminine or neuter pronoun. That ability alone is inadequate. Something mechanism must be in place to determine when it is appropriate and when it is appropriate to use a pronoun. The problem occurs when there is more than one message. For example, John received an 87 on his second midterm is fine, but the pronominal reference and referent are contained within the message expression. With two messages that do not communicate registers can generate either of the following: John received an 82 on his first midterm. John received an 87 on his second midterm., or He received an 82 on his first midterm. He received an 87 on his second midterm. or even, He received an 82 on his first midterm. John received an 87 on his second midterm. Messages must be coordinated, or an unacceptable document will result.

Foci are a design add-on to help control and coordinate messages. In the example above a message would use a pronominal reference for John only if it knows that John is already the SUBJECT focus. The focus mechanism makes it possible to reliably get John received an 82 on his first midterm. He received an 87 on his second midterm. Foci serve as something of a blackboard in the AI sense. A message can try to set the focus in order to let subsequent messages determine what is happening in the document. As a design add-on the implementation of foci is weak: Management is cumbersome for the author, and STRUCTED recognizes only SUBJECT, while CODEX recognizes SUBJECT and OBJECT. Foci should be defined by means of configuration files on a per-register basis. Associating foci with a message rather than with a message expression would simplify focus management.

6.1.3 Assessing Computational Registers

Three results mentioned in Section 6.1.1 support the success of computational registers as an approach worthy of continued effort. First, registers provide an organization that
can be applied to a variety of domains. Second, one tool, such as DEXTER, can be used to author registers in a variety of domains. DEXTER is not restricted to clinical medicine like IVORY [49], or to stock reports like ANA [37, 38]. Third, the quality of output exhibited in Chapters 4 and 5 is acceptable, especially for a first implementation. For instance, grammatical errors on output are possible, but generally due to an error in expression templates, and therefore able to be repaired on the fly. Texts have lexical and phrasal variability when desired, but the author can inhibit variability by defining only one expression per message. Registers can handle pronominalization and provide some means (foci) to control its use. Assuming the document-representation language allows for it, important points can be emphasized and multimedia included. Documents may be restructured based on data values. For example, in the Midterm Student Grade example, excessive missing work (more than 2) results in a more prominent location for the message pointing out missing work (Figures 5.1 and 5.5).

The various work cited in Section 1.4 gives some clues for different way to assess the success of registers as an approach to generating documents.

**Exhibiting Fluency**

As described in Section 1.4.3, Kukich has set forth some criteria for judging fluency in generated texts. DEXTER's registers avoid many fluency defects and acquire many fluency skills by relying on the register author to write expression templates. Within a message, any fluency problem such as appropriate use of pronouns, consistent verb tense, reasonable sentence length, parallelism, ellipsis and use of hyponyms is manageable. For the most part, sentence-level register output is as fluent as the author wants to make it, and only a little extra effort is required to achieve or better the fluency demonstrated in the examples of Chapters 4 and 5.

Fluency problems can occur between templates, so that redundant information, be it locative, temporal or otherwise, may appear. Pronouns may be over-used or under-used; ellipsis and aggregation may not be handled well. Foci are a first step toward improving inter-message communication in registers. In practice, a little care allows these problems to be resolved or avoided. A register-authoring tool can help by allowing the author to experiment with the register in the authoring environment, as DEXTER does. Software tools
that check grammar and check controlled language compliance have been deployed, for example EasyEnglish [5]. Existence of such tools raises the possibility of integrating similar tools, but capable of working with templates, into a register authoring system.

Meeting NLG goals

In Section 1.4.2, following Hovy and Reiter, I list some weaknesses of NLG systems. Registers do not solve any of the following problems: lack of general-purpose sentence-generation software, lack of a theory guiding lexical selection, sentence planning, discourse structure and choice between alternative generation strategies.

My approach to these problems is two-fold. First in my own approach to registers, I rely on the author’s ability to write templates. Humans are uniquely capable of general purpose sentence generation. At present we must use humans if that is what we want. Second, I designed registers to be modular, so that if research reveals a good algorithmic approach to sentence generation, discourse structure or so on, is developed, it can be incorporated into a register system. For example, expressions are clearly flexible templates in the sense defined by Pianta [62]. The register architecture can support using HTPL, mentioned in Section 1.4.2, to define expressions or to augment the procedural definition of expressions.

Side-stepping some of the issues raised by lack of underlying theories has two effects. It allows for a tool that works. It also provides fertile ground for study. Currently a good deal of research is devoted to studying various corpora of written text. The effect of having a body of registers could be helpful in establishing corpora of expressions, of discourse structures, and so on. Such study could prove illuminating in the search for theories to guide more sophisticated approaches to generation.

Registers do not side-step all NLG issues by relying on the register author, however. Registers directly attack some of the NLG limitations discussed in Section 1.4.2. The register architecture is not domain specific, nor is a register system domain-specific. Further, the programming and concepts involved with registers are familiar to most programmers. Register authoring is intuitive at a significant level (see the Section Compositionality below.) There may be no guiding theory of domain modeling, but there are candidate paradigms in programming. One is object orientation, which I have adopted as an approach to ontology definition.
My effort emphasizes using software tools to manage and build generation software. Software to manage software is sound engineering practice that seems to have been largely neglected in the NLG community. That neglect is not surprising, since the emphasis among NLG researchers has thus far been on systems dedicated to a specific domain.

Section 1.4.2 also notes several strengths and weaknesses of NLG systems versus traditional text generation systems. Registers have some features of both kinds of systems. DEXTER registers exhibit reasonable maintainability despite the use of templates. Deriving the Pulsar_Adm register from the Pulsar_MOTD register (Chapter 5) demonstrates adaptation to changing requirements. Smaller-scale changes in requirements can be dealt with more easily. For instance, one condition of one field rule would change to let the Mid Term Student Grade register recognize that the number of missing assignments is worthy of extra attention and increase its prominence in the final document. Expressions can be altered on the fly, and alternative document structures are defined as new rule trees in STRUCTED. Both the Pulsar_Adm and the Reg_Tutor register demonstrate that tables and graphics can be included in register-generated documents.

Meeting Communicative Goals

In Section 1.4.4 I describe Goffman’s concept of system and ritual constraints. Registers are designed to meet ritual constraints. For instance, Pulsar output is directed differently toward users and toward administrators. Different message expressions with different words and syntactic forms can be used in different registers, even if the registers share the same messages specifications.

In principle, system constraints should be satisfied by all communication, and registers can satisfy Goffman’s system constraints interpreted for electronic documents. I say “can” rather than “do” because, as usual, the burden falls naturally and largely on the register author. The relevant constraints are: open/close signaling introducing and abandoning topics, backchannel signaling incorporating non-verbal (non-textual) means of communication, turnover signaling indicating an exchange of communicative roles, marking digressions by means of bracket signals, and acoustic understandability and interpretability, meaning documents must be understood physically and semantically.

The development of an explicit document structure, the inclusion of context and focus information, and the ability to recognize applicability of alternative expressions of con-
stiuents, enable the register author to manage openings and closings of topics fluently. The definition of expressions, again combined with context information, allows the author to use non-verbal textual techniques such as emphasizing text. Another form of textual back-channel signal is alteration of the document structure (Section 5.1). A register author can write constituents bound to unexpected values. The author can use those constituents, and the rules that assert them, to anticipate questions from the document reader. A register understands expected input data values. Using a constituent to anticipate and answer a question is a kind of turnover signal. It pretends the user has been given the chance to ask a question, anticipates the question, and answers it.

Registers can clearly mark digressions with linguistic cues, or by creating linked documents. Links can be used to represent a digressive shift in a document. The usual forms of textual bracket signals are also available to the register author. For instance, a response to an anticipated question about an abnormal value may be marked by the strings The value for $X$ deserves notice. ... and Moving on from $X$, .... A register that implements data-carrying or includes vocabulary links allows the reader to issue a preempt signal by allowing the user to stop reading the current document to investigate the original data or the meaning of a word or phrase.

Finally, registers are *acoustically understandable and interpretable*. The analog of "Acoustically understandable" for documents is visual readability. The registrar screen shown in Example 1.1 illustrates a data presentation that is physically hard to read. Registers format text nicely and control volumes of data. There are several analogs of "interpretable" in documents. First, generated documents must be grammatically coherent, if not grammatically perfect. Second, a register outputs a document that *accommodates*; the tenor uses language appropriate to the reader. Accommodation serves as a metaphor motivating registers: If people learn to quickly accommodate, or match their language to their environments, then software should be as courteous and effective.

Goffman's system constraints also incorporate the four Gricean norms of relevance, quantity, truthfulness and clarity. Registers rely heavily on the register author to adhere to these norms, but registers implicitly support the author in doing so. Relevance, interpreted for documents, requires that each statement be meaningful in the context of the document. Extraneous matter should be filtered. The clarity norm reinforces that statements be meaningful in context. Statements should occur in a logical progression, so that the logical point of attachment of each statement to the whole document is approximated by the physical
location of the statement. Meaning should not be obscured by abnormal volume. The right level of terseness or verbosity is required by the norm of quantity. A statement appears in a document generated by a register because the field rules select the data embedded in the statement as pertinent, the content rules bind the data to a defined constituent, the structure rules select the constituent for expression and determine its place in the overall document, and the expression rendered has been checked for applicability given the data, context and focus of the document. The only way any of the norms just mentioned can be violated is if the author writes a bad register. Explicit support of these norms is another matter that requires further research, including a more precise interpretation of the norms. Registers also support the truthfulness norm. For a document, truthfulness incorporates the concept of user trust. Explicit support comes from the design of vocabulary and data-carrying to engender user trust in the generated document. Implicitly, domain-expert register authors are less likely to err and more likely to take care to be accurate. Therefore, by defining document generators so that domain experts are the authors, registers further support trust and truthfulness.

Compositional Plausibility

In Section 2.1 I adapt Reiter's "psycholinguistic plausibility" concept and claim registers are "psychocompositionally plausible" [67]. The mechanism of human speech formation, or even sentence formation, may be problematic. However, registers deal with written documents. What matters for register authoring is therefore compositional plausibility. A document generation approach is "compositionally plausible" if the generation plan follows any approach commonly taken by human authors. Compositional plausibility is important for the comfort, confidence and success of register authors, especially if less technical register authors are to be employed.

Composition is the art of constructing a document. It is taught in elementary, middle and high school as well as in college. To see that registers are compositionally plausible it is sufficient to observe that the register architecture parallels a standard approach to composition taught in schools. Most people will at least be familiar with the approach. A summary from standard student references on composition suffices to outline a usual approach to composition.

In The Lively Art of Writing Payne describes this approach to composition [61]:

1. take inventory
2. ask questions
3. look for relationships
4. ask yes-no questions
5. qualify

In *Student's Guide for writing college papers*, Turabian describes these steps to formulating a document (paraphrased) [74]:

1. accumulate largely unrelated facts and ideas
2. jot down all the ideas you have about what you know
3. find a logical order, recognizing incompleteness and irrelevancy
4. write the document

Neither of these organizational approaches is wholly isomorphic to computational registers. Yet registers are intuitively and readily interpreted in these terms:

1. **Field**: Examine the data by asking questions, accumulating relevant facts as independent entities.

2. **Mode-Content**: Ask questions that determine relationships among facts, noting all you might be able to say.

3. **Mode-Structure**: Given what you know and what you want to accomplish, decide the order of things to say, and which things to not say at all.

4. **Tenor**: Say what you have to say in the order indicated using appropriate language.

Some of the terminology of registers or the mechanism of the implementation might be unfamiliar to potential register authors. The overall organization, however, should be familiar to anyone who has studied composition in school. The familiarity of the organization should help register authors both learn the concept of a computational register quickly and feel comfortable using the approach.
The role of field processing in plausibility. Registers are designed to allow the use of templates in the tenor. Writing a grammar to express sentences requires technical training. Writing a template asks the meta-author to make a small abstraction based on what he already does routinely. Using templates for expression lends further compositional plausibility to my approach to registers. However, writing a template that is flexible enough to handle reasonable variance when working directly with raw data is difficult. Field processing effects a data transformation so that templates need not access raw data directly. Instead, the field re-represents the data so that the data is in a form amenable to expression in the given document. The result of field processing includes information about the data, such as which values are significant, and the grammatic attributes of values. This kind of information, combined with a task-appropriate representation of the data in the internal scheme, make writing a template expression relatively easy. The expressions of CODEX (Section 4.2.4) are templates that can respond to the data. Achieving the same degree of responsiveness in a template dealing directly with undescribed raw data would require the template to encode many more decisions. Templates that use the internal scheme representation of a register ask only the questions an author asks as he writes, for instance instance: “should a pronoun be used?”, or “what determiner is needed?”. Field processing is critical to making templates flexible while keeping them easy to use.

6.2 Future Work

There are two separate questions to think about for the future. First, what can be done to make registers work better and to better assess how they work? Second, what benefit can registers provide?

6.2.1 Future work with registers

With more experience building and using registers, and with a second, more robust, register-authoring tool, it will be possible to try to measure the utility of registers. How can we measure the benefit of having data presented by a register-generated document? It would be nice if we could say clearly that having these documents is of X degree of benefit in helping users of type Y understand the data. We might be motivated to pursue the task with more vigor, or even to drop the effort. However, the only way to determine the value of X is to have documents to use. Registers managed by a second-generation register
system can produce test cases. The result will be a measure of the utility of registers, but it will also have implications for the utility of generating documents. Solid, quantified evidence of the utility of generating documents is needed.

It will also be important to see how the restrictions registers place on input data can be relaxed. What is required to generate documents from less structured data? What are the problems in building registers that take data from more than one input source? Can registers be scaled up to larger documents? What is required of a register system, of input data, of a domain, in order to use registers to generate longer, more complex documents?

### 6.2.2 How to use registers

I see five fundamental ways to use registers. First, there are areas where applications can be developed. These areas are consistent with Hovy's suggestions for applying NLG technology as cited in Section 1.4.2: database content display, summarization of database content, and building output modules for expert systems and other software tools. The central focus of these tools is not presentation, but they can produce structured data. Likely application domains include clinical medicine, where clear norms of presentation exist, computer based examination, where adherence to ritual constraints is necessary for the comfort of the examinee, and all forms of personnel management, including both students and employees. The other uses are all research areas.

Information overload is a well acknowledged problem. The World Wide Web and electronic documents are a significant source of the problem [43]. The current generation of search engines yields voluminous results of variable accuracy. For this situation to improve, one of several things must happen: People must become better at handling massive quantities of information, there must be less information, search engines must become smarter, or documents must become more search-engine-friendly. Neither of the first two is likely likely to occur. Numerous efforts are underway to make search engines smarter and better able to filter the flood of electronic documents (for example [12, 41, 64]). It is possible to interpret DEXTER registers as filtering agents for a database. Given the ability to process less structured data, or the tools to put web search results into a database, registers can become a more generally applicable form of filtering agent.

Hand-written documents are not search-engine-friendly. Documents can become more search-engine-friendly in a number of ways. One way is through the use of controlled
language [20, 53], and another way is through the use of meta information [7, 26]. In principle, registers built using a register system’s authoring tools can simultaneously support both of these means of making documents search-engine-friendly. Vocabularies and expressions can constitute controlled language. Semantic tags, first mentioned in Section 2.4.1, are a form of meta data that can support finding, categorizing, summarizing or even extracting answers from documents. Therefore, registers with vocabulary management and semantic tagging (as in Section 2.4.1) merit study.

By design registers can target any of a number of pre-determined document representation languages, although DEXTER only supports HTML and plain text. Thus far I have found limitations in my registers due to limitations of DEXTER or due to limitations of my approach to registers. However, it is reasonable to expect that if registers and register systems improve, the document-representation language itself may become a point of limitation. Registers may become a research tool to help us probe what a good document-representation language should include. A parallel no doubt lies in the history of programming languages. Presumably the problems writing programs with the first FORTRAN compilers had to do with limitations of the compiler and its implementation. Eventually algorithms were developed that indicated a need for different language features.

A related use is using registers to write computer code. Programming languages are formally defined, precise languages. If, therefore, a register can be used to generate acceptable natural language, could a similar approach be used for automatic-programming? The first obstacle is lack of a representation of programming problems that is broad, convenient, and supports formal queries.

Finally, registers may be of use to linguists. Currently, many analyses of text corpora are made, often in the study of language generation. A register, authored by a human, is a document definition one level up. Would the body of expressions a human author chooses to include in the tenor of a register form an interesting linguistic study? What would the rules the author uses to consciously determine the content and structure of the document, or the structure tree of the document itself, reveal about the nature and definition of documents and writing?

### 6.3 Conclusions

The goals I enumerate for the register architecture in Section 2.1 are:
1. The approach must support building documents that are suitable for presentation of data.

2. The approach cannot be overly restrictive.

3. Standard methods of computer science must be sufficient to implement the approach.

4. The approach must be as straightforward as possible so that domain experts can help define document generators in their own domain.

5. The approach must lend itself to systematic management of generation software, so that software systems can be created to help design, implement and maintain document generation programs.

Computational registers as implemented in DEXTER satisfy these goals at a basic level. The documents I have shown provide evidence that goal 1 is satisfied. There are problems with expression due both to DEXTER and to the design of registers. However, the documents exhibited show a reasonable fluency. The goal documents of Section are approximated by the resulting documents. Further, registers satisfy communicative goals.

Goal number 2 is generally satisfied. I do not mean that registers can be used anywhere for any document-generation needs. DEXTER registers are restricted at present to dealing with relatively short documents, represented in HTML, and based on data stored in a simple relational database. Within these constraints, I have shown that registers built by one register system work across a good variety of domains. DEXTER registers have addressed: grade reports, description of people and their pets, descriptions of basic facts about universities and the status of a Unix network as reported by another software tool.

DEXTER is implemented in Java. It builds registers by building rules as Java classes. Since the classes are in most cases built after the tools that execute them, it is necessary to be able to construct objects from string descriptions. DEXTER does this through the java.lang.reflect API. DEXTER's technology consists primarily of rules, string manipulation and organization. The organization, although composed of four significant components, is not complex, and has a clear parallel in human authoring. Therefore, DEXTER is a register system that shows that computational registers can be implemented using only tools familiar to a great majority of computer programmers. The linguistic
knowledge required to write a CODEX expression is either high-school level, or is obtainable via a dialog format. Thus goal 3 is satisfied.

Goal 4 presents a difficult problem. Computational registers support a stratified, team authoring approach. The register field is somewhat technical. Although this may be improved by a better DEXTER, the problem of mapping data from one paradigm, say relational databases, to another such as classes of an object-oriented programming language, is an area of ongoing research [3, 57]. Incorporating presentation information into the result of that mapping only makes the problem more difficult. It is therefore likely that register-field authoring will continue to require some technical (database and programming) skill. However, beyond the field, the level of technical resources required diminishes.

For constituent definition, content determination, structure determination and expression definition, substantially less technical skill is required, even in an early tool like DEXTER. A better system should require less skill to use. DEXTER takes a good step toward realizing goal 4 above, and shows that it is feasible for computational register systems to move further toward that important goal.

Regarding goal 5, DEXTER implements most of a register system. There is work to be done to improve DEXTER, especially with respect to altering the design so that registers may share resources, and with respect to getting DEXTER to help a register author work with entire sets of rules. Various registers can be written using DEXTER, even though those registers differ with respect to domain. Those registers may have expressions modified on the fly, and structure or content selection may be modified without requiring change in earlier components, unless, for example, new information is required. DEXTER is a good first step in systematic register management.

6.3.1 Summary of Contribution

I have proposed a method of document generation called “computational registers”. I have built a system, DEXTER, which shows that registers can be successfully implemented. The register approach is generally consistent with the approach of many researchers in NLG in treating content determination and document planning as computational tasks. At the same time, the approach supports the simplicity and familiarity of using templates to express data. The approach is unique in several ways. First, content determination and structural planning are carried out sequentially rather than simultaneously. Second,
content planning itself is broken down into data re-representation followed by content selection. Third, no advanced linguistic knowledge is required to understand the approach. Fourth, computational registers are intended to be, and appear to be, applicable in a variety of domains. What registers can be written is restricted not by domain, but by the original data representation. I believe it will be easier to relax these constraints with experience over time than to try to redesign the approach to take into account new demands placed by new content domains. Finally, DEXTER shows computational registers can be managed by software. Software tools can be used to write, test, and maintain registers in different domains.
Bibliography


[33] Illustra Inc. Illustra multimedia information system user's guide, 1994. Illustra is now known as Informix Universal Server.


Vita

Date and place of birth: August 20, 1959, in Lexington, Kentucky

Education:
1987 B.A., English University of Kentucky, Lexington
1984 A.D., Nursing Lexington Technical Institute, Lexington, KY
1978 Diploma Henry Clay Sr. High School, Lexington, KY

Professional Positions:
Instructor, Department of Computer Science, University of Kentucky, Lexington, 1998 - present.

Graduate Research Assistant, Department of Computer Science, University of Kentucky, Lexington, 1996 - 1998.

Graduate Teaching Assistant, Department of Computer Science, University of Kentucky, Lexington, 1992 - 1996.

Graduate Research Assistant, Department of Computer Science, University of Kentucky, Lexington, 1992 - 1994.

**Professional Publications:**

(With V. Marek and M. Truszczyński) *Intelligent Computation of Presentation Documents*


(With V. Marek) *Toward Intelligent Representation of Database Content*,

__________________________

(Joseph Dowell Oldham)
Acknowledgments

I could not have succeeded in this effort without the assistance of three faculty members. First, Professor Victor Marek has shown unending patience, offered unfailing encouragement, and provided innumerable insights in guiding my research efforts. Professors Mirek Truszczynski and Raphael Finkel have provided insight and encouragement and given very generously of their time. I offer them all sincere thanks. I also owe thanks for various assistance and support to Professors Greg Stump, Tony Baxter and Judy Goldsmith, and to Diane Mier, Martha Wells, Jane Spanyer and Paul Linton. Numerous of my fellow graduate students have contributed as well: Debby Keen, Tom Kay, Jonathan Edwards, Carol Hannahs, Billie Sue Chafins, Tony Borchers, Robert Adams, Artur Mikitiuk and Angel Janevski. My deepest and most humble thanks go to my wife, Dianna Howard, and to our son, Will Oldham. They have been the foundation and the inspiration at every step.

This work was supported by USARO contract DAAH-04-96-1-0398.
Table of Contents

Acknowledgments iii
List of Tables vii
List of Figures viii

1 Introduction 1
   1.1 Motivation .................................................. 2
   1.2 Problem Statement .......................................... 8
   1.3 Sketch of the Approach .................................... 10
   1.4 Related Work ................................................. 11
       1.4.1 Terminology ........................................... 11
       1.4.2 Overview of NLG ...................................... 12
       1.4.3 A Few Generation Systems .......................... 14
       1.4.4 Communicative Goals ................................ 16
       1.4.5 Plausibility ........................................... 17
   1.5 Summary ....................................................... 17

2 The Computational Register Architecture 19
   2.1 Architectural Goals ......................................... 19
   2.2 Field .......................................................... 22
       2.2.1 Constraints in Field Processing .................... 24
   2.3 Mode .......................................................... 25
       2.3.1 Mode-Content ........................................ 25
       2.3.2 Structure .............................................. 27
   2.4 Tenor .......................................................... 30
2.4.1 Embedded Descriptions of Documents ............... 32
2.5 Summary ............................................. 33

3 Design Decisions ................................. 35
3.1 Registers are Rule-based ............................ 35
3.2 Procedural Templates to Express Constituents .......... 36
3.3 Context and Focus ................................. 38
3.4 General Approach, Database and Ontology ............... 39
3.5 Field ................................................. 39
   3.5.1 Rules and Queries ............................... 39
   3.5.2 Taxonomy of Field Rules ......................... 42
   3.5.3 Vocabulary ..................................... 43
3.6 Mode ................................................. 44
   3.6.1 Content Rules .................................. 44
   3.6.2 Structure Rules .................................. 47
3.7 Templates and Expressions in the Tenor .................. 49
3.8 A Register System: Management via Editors ............. 50
   3.8.1 Authoring one register ............................ 50
   3.8.2 Managing Several Registers ......................... 52
3.9 Issues and Restrictions ................................ 53

4 DEXTER: An Implementation ........................ 55
4.1 Overview and Platform Issues ........................ 55
   4.1.1 Platform ....................................... 55
   4.1.2 Overview ...................................... 56
4.2 Building a Register .................................. 57
   4.2.1 Pre-processing: data, corpus, queries, classes .......... 57
   4.2.2 Declaring a Register with ADMIN ................. 70
   4.2.3 Building the Field with FRED ....................... 71
   4.2.4 Determining content: CONED, CODEX, MORCONED . . . 93
   4.2.5 Determining Structure: PRESED, CODEX, STRUCTED .. 113
   4.2.6 Revisiting FRED for an Iterative rule and a Take-first rule . 124
4.3 Reg_Tutor Completed ................................ 128
5 More Registers
  5.1 Midterm Student Grade ............................................. 132
  5.2 University ............................................................ 135
  5.3 Pulsar ................................................................. 137

6 Evaluation, Conclusions and Future Work .......................... 142
  6.1 Evaluation ............................................................. 142
  6.1.1 Evaluating DEXTER and Computational Registers ........ 142
  6.1.2 Implementation issues ............................................ 143
  6.1.3 Assessing Computational Registers ............................ 147
  6.2 Future Work ........................................................... 154
  6.2.1 Future work with registers ..................................... 154
  6.2.2 How to use registers ............................................ 155
  6.3 Conclusions ............................................................ 156
  6.3.1 Summary of Contribution ....................................... 158

Bibliography ..................................................................... 160

Vita ................................................................................. 168
List of Tables

5.1 Re-use of resources in deriving Pulsar_Adm from Pulsar_MOTD. . . . . . . 139
5.2 Time in hours developing Pulsar_MOTD, deriving Pulsar_Adm. . . . . . . 139
List of Figures

1.1 Computational registers ............................................. 2
1.2 Register internals .................................................. 10
1.3 More details of register processing ............................... 11
2.1 Schematic of register processing. ................................. 33
2.2 Register details, components and processing ................. 34
4.1 Schematic of DEXTER as implemented. .......................... 56
4.2 FRED, showing built-in objects in the meta folder, query results in the working folder. .......................................... 74
4.3 Defining FoundTarget ............................................... 76
4.4 Defining Target .................................................... 77
4.5 Initializing FoundTarget ............................................ 78
4.6 Creating the verifyTarget singleton rule on query getPerson. .... 79
4.7 The assignment of verifyTargetRule, showing FRED’s edit tools palette. ...................................................... 81
4.8 FRED’s BUILD Menu ................................................ 83
4.9 Simulating a register’s run time arguments in FRED .......... 84
4.10 Detailed view of two objects after verifyTargetRule is executed on Victor March ........................................... 85
4.11 FRED’s fixed templating facility. .................................. 86
4.12 Conjuncts appear one per line .................................... 90
4.13 Executing a list of field rules in FRED ........................... 93
4.14 Reg_Tutor’s sentence_one fixed template filled after list execution. ...................................................... 94
4.15 CONED defining IDAgeHtWt with parameter Target in Reg_Tutor ..................................................... 96
4.16 CONED ready to accept message IDAgeHtWt. ................. 97
4.17 CONED with two constituents defined and built. .............. 98
4.18 CODEX showing Message Description tab. ..................... 99
4.19 CODEX showing gender choice on Person.IAge. .............. 104
4.20 MORCONED defining KNOW for FirstIdentification ......... 110
4.21 MORCONED, binding Target to Person .......................... 111
4.22 MORCONED shows results after executing FirstIdentification ..................................................... 112
4.23 IdentifySubject defined in STRUCTED .......................... 120
4.24 Beginning the rule tree ............................................. 121
4.25 Three rules in the tree. .......................... 122
4.26 Same tree as 4.25, drawn. .......................... 122
4.27 Completed tree. ................................. 123
4.28 Same tree as 4.27, drawn. .......................... 123
4.29 Messages following execution of tree Reg_Tutor_T1. .......... 124
4.30 Results following execution of tree Reg_Tutor_T1. .......... 124
4.31 The HTML document displayed in the Tenor Window. .......... 125
4.32 The FindPets iterative rule action list. .......................... 127
4.33 About Victor March, shown in the completed register. .......... 128
4.34 About Joseph Oldman, shown in the completed register. .......... 128
4.35 About Miroslaw Szczygielski, shown in the completed register. .......... 128
4.36 About Beth Silverstein, shown in the completed register. .......... 129
4.37 About Natalia Sturek, shown in the completed register. Notice “She no pets.” .......................... 129
4.38 About Brenda Chan, shown in the completed register. .......... 130
4.39 Final rule tree for Reg_Tutor. .......................... 131

5.1 The Goal_Tree rule tree of the Midterm Student Grade register. .... 133
5.2 The Simple_MTG_Report rule tree of the Midterm Student Grade register. .... 133
5.3 Output using Goal_Tree in Figure 5.1. .......................... 134
5.4 Output using Simple_MTG_Report in Figure 5.2. .......................... 134
5.5 Reordering messages when too many assignments are missing. .......... 135
5.6 University register output, same data as Figure 5.7. ............ 136
5.7 University register output, same data as Figure 5.6. ............ 136
5.8 Pulsar_MOTD output on the same data as Figure 5.9. ............ 138
5.9 Pulsar_Adm output on the same data as Figure 5.8. ............ 141
ABSTRACT OF DISSERTATION

Joseph Dowell Oldham

The Graduate School
University of Kentucky
2000
Generating Documents by means of Computational Registers

ABSTRACT OF DISSERTATION

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the University of Kentucky

By
Joseph Dowell Oldham
Lexington, Kentucky

Director: Dr. Victor Marek, Professor of Computer Science
Lexington, Kentucky
2000
ABSTRACT OF DISSERTATION

Generating Documents by means of Computational Registers

Software is often capable of efficiently storing and managing data on computers. However, even software systems that store and manage data efficiently often do an inadequate job of presenting data to users. A prototypical example is the display of raw data in the tabular results of SQL queries. Users may need a presentation that is sensitive to data values and sensitive to domain conventions. One way to enhance presentation is to generate documents that correctly convey the data to users, taking into account the needs of the user and the values in the data.

I have designed and implemented a software approach to generating human-readable documents in a variety of domains. The software to generate a document is called a computational register, or “register” for short. A register system is a software package for authoring and managing individual registers. Registers generating documents in various domains may be managed by one register system. In this thesis I describe computational registers at an architectural level and discuss registers as implemented in DEXTER, my register system. Input to DEXTER registers is a set of SQL query results. DEXTER registers use a rule-based approach to create a document outline from the input. A register creates the output document by using flexible templates to express the document outline.

The register approach is unique in several ways. Content determination and structural planning are carried out sequentially rather than simultaneously. Content planning itself is broken down into data re-representation followed by content selection. No advanced linguistic knowledge is required to understand the approach. Register authoring follows
a course very similar to writing a single document. The internal data representation and content planning steps allow registers to use flexible templates, rather than more abstract grammar-based approaches, to render the final document. Computational registers are applicable in a variety of domains. What registers can be written is restricted not by domain, but by the original data representation. Finally, DEXTER shows that a single software suite can assist in authoring and managing a variety of registers.

(Joseph Dowell Oldham)

(Date)
Generating Documents by means of Computational Registers

By

Joseph Dowell Oldham

(Director of Dissertation)

(Director of Graduate Studies)

(Date)
RULES FOR THE USE OF DISSERTATIONS

Unpublished dissertations submitted for the Doctor’s degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the dissertation in whole or in part requires also the consent of the Dean of The Graduate School of the University of Kentucky.

A library which borrows this dissertation for use by its patrons is expected to secure the signature of each user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Generating Documents by means of Computational Registers

DISertation

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the University of Kentucky

By
Joseph Dowell Oldham
Lexington, Kentucky

Director: Dr. Victor Marek, Professor of Computer Science
Lexington, Kentucky

2000