A Logic Data Model for the Machine Representation of Knowledge

Department of Computer Science
University of Waterloo
Waterloo, Ontario

Randy Goebel

Technical Report CS-86-07

A logic data model for the machine representation of knowledge

by

Randy Goebel

June 1985

Abstract

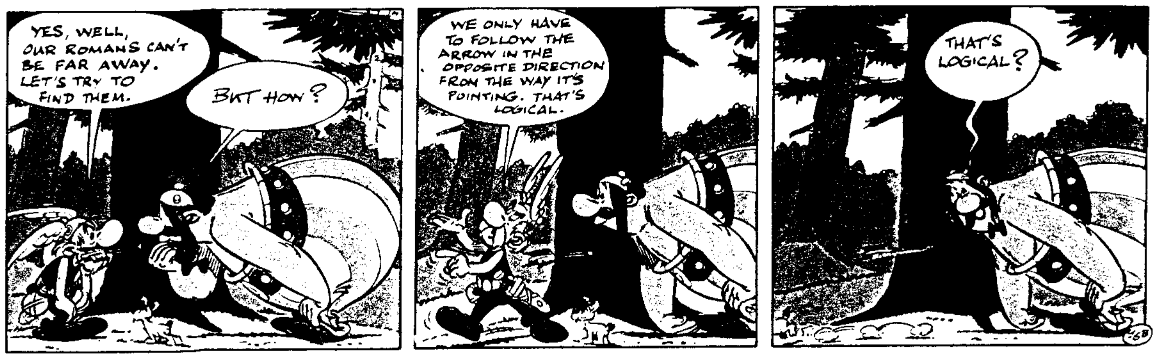
DLOG is a logic-based data model developed to show how logic-programming can combine contributions of Data Base Management (DBM) and Artificial Intelligence (AI). The DLOG data model is based on a logical formulation that is a superset of the relation data model [Reiter83], and uses Bowen and Kowalski's notion of an amalgamated meta and object language [Bowen82] to describe the relationship between data model objects. The DLOG specification includes a language syntax, a proof (or query evaluation) procedure, a description of the language's semantics, and a specification of the relationships between assertions, queries, and application databases.

DLOG's basic data description language is the Horn clause subset of first order logic [Kowalski79, Kowalski81], together with embedded descriptive terms and non-Horn integrity constraints. The embedded terms are motivated by Artificial Intelligence representation language ideas, specifically, the descriptive terms of the KRL language [Bobrow77]. A similar facility based on logical descriptions is provided in DLOG. The DLOG language permits the use of definite and indefinite descriptions of individuals and sets in both queries and assertions.

The meaning of DLOG's extended language is specified by writing Horn clauses that describe the relation between the basic language and the extensions. The experimental implementation is the appropriate Prolog program derived from that specification.

The DLOG implementation relies on an extension to the standard Prolog proof procedure. This includes a "unification" procedure that matches embedded terms by recursively invoking the DLOG proof procedure (cf. LOGLISP [Robinson82]). The experimental system includes logic-based implementations of traditional database facilities (e.g., transactions, integrity constraints, data dictionaries, data manipulation language facilities), and an idea for using logic as the basis for heuristic interpretation of queries. This heuristic uses a notion of partial match or sub-proof to produce assumptions under which plausible query answers can be derived.

The experimental DLOG database (or "knowledge base") management system is exercised by describing an undergraduate degree program. The example application database is a description of the Bachelor of Computer Science degree requirements at The University of British Columbia. This application demonstrates how DLOG's embedded terms provide a concise description of degree program knowledge, and how that knowledge is used to specify student programs, and select program options.


YES, WELL, OUR ROMANS CAN'T BE FAR AWAY. LET'S TRY TO FIND THEM.
BUT HOW?
WE ONLY HAVE TO FOLLOW THE ARROW IN THE OPPOSITE DIRECTION FROM THE WAY IT'S POINTING. THAT'S LOGICAL.
THAT'S LOGICAL?

—Goscinny and Uderzoo, *Asterix and the Great Crossing*

Preface

This report is a slightly revised version of doctoral dissertation accepted at The University of British Columbia in October of 1985.

I believe that the research reported here is about knowledge representation for Artificial Intelligence. The basic premise is that viewing a representation language as a logic provides a methodology for investigating an important aspect of representation: the relation between symbols of a formal language and the real or artificial world that such symbols represent. The support given this premise is the design, implementation, and application of a representation language called DLOG.

Many have strongly criticized the role of logic in AI. In response, others have strongly defended it. These extreme critics and defendants have often made claims that border on the extra-galactic. The whole point of using logic as a tool is the expedience offered by being precise about the relationship between symbols and the worlds that they describe. Having committed myself to a desire for precision, it's important to admit that the work reported here offers no new theorems about representation, nor does it go so far as to prove that the implementation retains the fidelity of the DLOG representation language.

The semantics of DLOG are described in two different ways. The first uses first order Tarksian semantics, which provides an account for most of the DLOG language but falls short of capturing the intended semantics of certain complex terms. However, the intended semantics of these terms can be described at the meta level in a straightforward way, so that an implementation of a DLOG proof system can be defined in terms of a Prolog meta program. While the implementation described is relatively simple, it remains true that the complete DLOG semantics depends on an amalgamation of meta and object language such as proposed by Bowen and Kowalski.

A second approach to the description of DLOG's semantics uses a more powerful system of logic: Montague's second order intensional logic. The terms which were problematic for the first order semantics are easily seen to correspond to Montague's notion of "obligation," and their manipulation by a proof system can still be described, informally, by a Prolog meta program. However, as a general second order intensional theorem prover has not been constructed, the use of a second order intensional semantics leaves another formal gap between the language's semantics and the prototype implementation.

Despite this lack of a correctness proof for the prototype DLOG interpreter, the instances in which meta programming techniques substitute for a more general proof procedure are few and are clearly specified. In fact, the style in which the non-first order extensions are manipulated is claimed to be as important as the terms themselves.

As this dissertation makes several claims about different aspects about the problem of designing, building and using a knowledge representation system, the reader may want to first read [Goebel85b, Goebel85a] which are rather more brief and to the point. These two papers provide a summary of two different aspects of this research, and will provide the reader with many of the points made in this document.

Finally, a word of warning to those who might attempt to recover the basic ideas from reading the implementation code in the appendices. The program given was written during the formative years of my logic programming experience; it is, in most cases, *not* exemplary Prolog programming. The prototype was sufficiently robust to manage the correct interpretation of the complete undergraduate program, and to produce the examples used in chapter two. However, if you don't understand the basic ideas from the descriptions given in the text, you probably won't get them by skipping to the implementation either.

Preface references

[Goebel85a] R. Goebel (1985), Interpreting descriptions in a Prolog-based knowledge representation system, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, August 16-18, UCLA, Los Angeles, California, 711-716.

[Goebel85b] R.G. Goebel (1985), The design and implementation of DLOG, a Prolog-based knowledge representation system, *New Generation Computing* **3**(4), 385-401.

Contents

Tables

Figures

Acknowledgements

I acknowledge my supervisor, Ray Reiter, for demonstrating that I am not as clever as I thought; for his integrity in holding to his views; for pushing me to research horizons that I didn't know existed; and for making me psychologically tough enough to endure anyone's criticism.

To the various members of my committee for their support over the years, especially Bary Pollack for his encouraging compliments; Alan Mackworth, for periodically reminding me that this thing is a human endeavour; to Richard Robinson, for his technical skill with mathematics and his personal skill at helping me verify the accuracy of my intuition; and to Harvey Abramson and Richard Rosenberg, for painfully wading through drafts much less clear than this one.

To the various others who provided objectivity when I could find none: to Paul Gilmore, for enduring my illogical fancies and pointing me to Hilbert; to John Mylopoulos, who pointed out that actually implementing something was important too; to Len Schubert, for his unfailing support of confindence; to Nick Cercone and Gord McCalla, who prevented me from taking myself too seriously; to some of my fellow computer science students John Demco, Roger Browse, Jay Glicksman, Jan Mulder, Jim Little, and Iggy Mercer, who collectively shared both my enthusiasm and distress.

To my colleagues at the University of Waterloo: to David Poole, for labouring through previous drafts and finding clarity amidst the symbols; to Maarten van Emden, for his confidence in my motivation; to Romas Aleliunas, for repeatedly placing mathematics in an appropriate context; to John Brzozowski, for his confidence against all odds; and to John Beatty, Ian Munro, and Kelly Booth for turning the phrase "Is it done yet?" into a normal everyday salutation.

Finally, to Lanie, Kari, and Jodi, for enduring.

Chapter 1
Introduction

1.1. Databases and knowledge bases

Any research that addresses the use of information by machine will encounter the terms *database* and *knowledge base.* One of the terms must be consistently used, even though the choice is rarely discussed or justified. For example, "knowledge" is more pretentious than "data," and may be selected for no other reason.

The assumptions evoked by the two terms serve as the basis from which a consolidated view can be synthesized. The motivation and accomplishments of this research are best understood by reviewing some of these assumptions.

1.1.1. Distinguishing knowledge bases and databases

To begin, consider a comparison between a database system and knowledge base system. Initially these are simply two programs that implement various facilities for capturing and manipulating some form of machine-storable information.

The first tangible distinction is in the motivation for constructing the systems. For example, Artificial Intelligence (AI) knowledge bases are viewed as repositories for an intelligent program's domain knowledge. In a sense, the system is anthropomorphized: the program is an individual whose knowledge base forms its foundation for understanding and behaving in its environment. In contrast, the traditional Database Management (DBM) conceives a database as a collection of information to which uniform access is provided, usually by enforcing a uniform storage format, and by providing a uniform retrieval mechanism. Depending on the perceiver, this often vague distinction can evaporate or grow (e.g., [Wong77]).[1]

[1] See also the comments of E.F. Codd and I.P. Goldstein [Brodie81, pp. 88-89].

However, admitting these distinct motivations suggests more tangible distinguishing criteria: (i) kinds of information, (ii) volume of information, (iii) inference capabilities, and (iv) administrative capabilities.

1.1.1.1. Kinds of Information

Some have argued that the information stored in databases and knowledge bases is quite different (e.g., [Abrial74, Wong77]). One contention is that knowledge bases are intended to capture abstract information, e.g., "*all* cats are animals," in addition to the concrete information typical of a database, e.g., "Joe is a cat." When limited to this notion, recent debates have substituted "structured versus unstructured" for the more vague "concrete versus abstract" distinction.[2]

Here the reluctant consensus is that databases hold large quantities of highly structured data, while knowledge bases contain less regularly structured information. Additional comments on structure argue that databases distinguish *conceptual schema* information from the database proper, but that knowledge bases incorporate both in a uniform way. (In chapter 2, we discuss a concept that can be used to verify these intuitions in a more precise way.)

1.1.1.2. Volume of Information

The relative volumes of information stored in databases and knowledge bases differ dramatically.[3] This is accounted for, in part, by noting that databases exist as real world applications, whereas knowledge bases embrace restricted and often contrived domains. But volume remains a weak distinguishing criterion since it so intimately depends on *how* information is encoded: sophisticated inference can buy economy of volume.

---

[2] E.g., see remarks in [Brodie81] especially pp. 17-18, and pp. 40-42.

[3] E.g., see S.N. Zilles comments in [Brodie81, p. 88].

1.1.1.3. Inferencing and information dynamics

Debates about kinds and volume of information often evolve to issues about how information is used. The knowledge base's emphasis on lower volumes of flexibly structured information implies the use of inferencing to recover implicit information. This contrasts with traditional databases, which store large volumes of rigidly structured information and use retrieval mechanisms based on that structure. Ultimately, the inferencing capabilities of a system depend on the structure of information retained, so the imprecise distinctions based on kind or volume of information can usually be absorbed by distinguishing inference methods.

The need for inference is partly determined by information dynamics: volatile collections of data require "soft structuring," i.e., relationships are left implicit and are recovered by an inference mechanism. In contrast, the relatively static conceptual schemas of traditional DBM can support "hard structuring:" implicit relationships can be computed and stored, rather than derived at retrieval time. For example, Bernstein [Berstein76] offers an algorithm for structuring a relational database according to a prespecified set of dependencies. These dependencies are static — any change would require database reorganization. Similar structuring ideas (e.g., [Armstrong80]) increase the efficiency of manipulation at the expense of flexibility in accommodating changes to general knowledge (i.e., the conceptual schema). In knowledge bases, improvements in efficiency gained by sophisticated structuring of specific facts is dependent on the corpus of general facts, but the general facts may change as quickly as the specific ones.

1.1.1.4. Administrative capabilities

One final distinguishing feature arises from the original assumption about alternative motivations. The focus is on how well administrative functions of a system are automated (e.g., determining the credibility of new information). For example, a database administrator is responsible for the integrity and accuracy of a database, but the anthropomorphic conception of a "knowledge-based" system evokes a view of a complete agent, responsible for its knowledge and beliefs within technological constraints. Specifically, a human data administrator might establish the credibility of potential updates by interrogating their originators; a knowledge based system might undertake to establish this credibility on its own — belief systems lie solely within AI.

There is some intersection in the area of integrity constraints, where inferencing capabilities determine the ease of enforcing such constraints. The issues revolve around the basic paradigm for maintaining consistency. In AI, the consistency issue is usually addressed in the logical paradigm, while DBM has concentrated on normal form theory.

1.2. DLOG: a logic-based data model

DLOG is a logic-based data model developed to show how logic-programming can combine contributions of DBM and AI. The DLOG data model is based on a logical formulation that is a superset of the relational data model(cf. [Reiter83]), and uses Bowen and Kowalski's idea for an amalgamated meta and object language [Bowen82] to describe relationships among data model objects. The DLOG specification includes a language syntax, a description of the language's semantics, a description of DLOG query evaluation as DLOG derivability, and a description of the relationship between assertions, queries, and application databases.

DLOG's basic data description language is the Horn clause subset of first order logic [Kowalski79, Kowalski81], together with embedded descriptive terms and non-Horn integrity constraints. The embedded terms are motivated by AI representation language ideas. In particular, the descriptive terms of the KRL language [Bobrow77b] have motivated a similar facility based on logical descriptions. The DLOG language permits the use of definite and indefinite descriptions of individuals and sets in both queries and assertions.

The meaning of DLOG's extended language is specified by logical assertions that relate the basic language and the extensions. The experimental implementation is the appropriate Prolog program derived from that specification.

The DLOG proof procedure relies on an extension of SLD resolution, the foundation of Prolog's proof procedure [Lloyd82]. Extensions include a "unification" procedure that matches embedded terms by recursively invoking the DLOG proof procedure (cf. LOGLISP [Robinson82b]). The extended unification can be viewed as a general SLD proof procedure with a new theory of equality [Emden84]. The experimental system includes logic-based implementations of traditional database facilities (e.g., transactions, integrity constraints, data dictionaries, data manipulation language facilities), and an idea for using logic as the basis for heuristic interpretation of queries. This heuristic uses the notion of partial-match of sub-proof to produce assumptions under which plausible query answers can be derived.

The experimental DLOG database (or "knowledge base") management system is exercised by describing an undergraduate degree program. The example application database is a description of the Bachelor of Science with a major in Computer Science degree requirements at The University of British Columbia. This application demonstrates how DLOG's embedded terms provide a concise description of degree program knowledge, and how that knowledge is used to specify student programs, and select program options.

1.3. Logic and databases

This section provides the historical background and major sources from which this research derives. The topic "logic databases" spans considerable breadth in the literature — mathematical logic, DBM, AI — and presents an intimidating horizon for the reviewer. The approach will be to divide the contributions into five divisions and acknowledge the important ideas in each of these. Each acknowledgement is further classified by

(1) recognition of historical results and ideas fundamental to the division,

(2) recent ideas central to the motivation and background for this research, and

(3) recent work similar enough to be acknowledged as supportive.

The reader is reminded that the phrase "logic databases" has been adopted as the name of a field that studies logic as a tool for DBM [Gallaire78, Gallaire81], even though the term is generally more inclusive. For example, logic programming is a major tool of the logic database community, but not necessarily synonymous with Prolog programming (cf. [Minker78, Weyrauch80, Robinson82a]).

1.3.1. Theorem-proving and logic programming

After the feasibility of first-order proof procedures had been demonstrated [Gilmore60, Davis60, Prawitz60], major interest in mechanized theorem-proving awaited the resolution inference method of Robinson [Robinson65]. Since Robinson's paper on the resolution inference rule, most efforts at mechanizing proof theory have focused on improving the efficiency of resolution (e.g., [Kowalski71, Reiter71]), although some work has been done with natural deduction systems (e.g., [Bledsoe77b, Weyrauch80, Haridi83]). The contribution of Green [Green69] is central to the use of theorem-proving in AI, as it demonstrated how resolution theorem-proving could be used for question-answering.

The development of logic programming as the top-down, depth-first interpretation of Horn clauses (e.g., [Kowalski79]), provides the basic implementation tool for this research. Subsequent work in combining meta and object level language in Prolog systems [Bowen82] provides a basis for specifying the relationship between DLOG assertions, queries, and application databases.

After constructing the first DLOG prototype, the work of Robinson [Robinson79] and Robinson and Sibert [Robinson82a, Robinson82b] provided both technical and moral support. Robinson's exposition of *exemplifications* and the semantics of Hilbert terms helped clarify both the construction and interpretation of DLOG's embedded descriptions.

1.3.2. Knowledge representation

The idea that a formal language is an appropriate tool for investigating machine representations of knowledge is usually attributed to McCarthy [McCarthy68], and the work by him and others of the same persuasion (e.g., [McCarthy69, Hayes74, Hayes77, McCarthy77, Reiter80]) provide important contributions. The work of Bobrow and Winograd [Bobrow77b], Brachman [Brachman79] and semantic network developers [Findler79], and Moore and Newell [Moore74] have helped show how formal logic is but a tool, and that intuition about plausible reasoning and informal knowledge is a necessary ingredient in the development of representation theories. McDermott's ideas [McDermott78a, McDermott78b] seem to mediate the two basic positions, not denying the importance of intuition, but demanding denotational interpretations of representation languages.

The most direct influence on this research comes from Bobrow and Winograd's KRL system [Bobrow77b], since the proof mechanism of DLOG is an attempt to achieve, in a logic-based system, a portion of their "mapping" style of reasoning.

Some other research (e.g., [Moore76, Schubert76, Attardi81]) use descriptions in various ways related to DLOG. Of these, Schubert's brief discussion,[4] is the most relevant since it proposes a method for interpreting descriptions whose presuppositions are not met. Moore [Moore76] proposes the use of descriptive terms to distinguish opaque and transparent contexts in natural language; he suggests a referential interpretation of descriptive terms, but no implementation is provided. The OMEGA system of Attardi and Simi [Attardi81] views all descriptive terms as indefinite descriptions of sets; while their interpretation of descriptions is different, the motivation seems related: to provide a logical interpretation of embedded terms.

1.3.3. Database Management

The concept of a data model is central to the idea behind this work, and owes a debt to many researchers in DBM (e.g., see [Fry76]). The relational model of Codd [Codd70, Codd82] has exhibited the virtues of data independence with great clarity, and has provided the basis for some impressive implementations [Stonebraker76, Stonebraker80, Astrahan76, Chamberlin81].

Wong and Mylopoulos [Wong77] were among the first to consider the relationship between AI and DBM, and the first to suggest dimensions for comparing them. Their later work has used the data model concept to formalize semantic network ideas in the TAXIS system [Mylopoulos80, Borgida81].

[4] [Schubert76, pp. 185-188]

1.3.4. Artificial Intelligence and databases

AI rarely considers "databases," but most often investigates the application and development of "knowledge bases." This viewpoint has focused attention on the performance of particular instances of knowledge-based systems, with less consideration for data independence. Davis' [Davis76] considerable effort in addressing the issues involved with building expert systems has spawned one of the few systems that emphasizes the data independence of knowledge base management tools [Melle81]. This work is evidence of the growing concern for data independence in AI, and demonstrates the need for further understanding of data independence in knowledge bases.

Another important facet of knowledge base research has concentrated on user interfaces, and the kind of knowledge needed for intelligent interaction (e.g., [Kaplan79, Mays81, Davidson82]). This aspect of knowledge base use is within the comprehensive proposal of Codd [Codd74], and the ambitious REL project of Thompson et al. [Thompson69]. The goal is to develop techniques for enriching the standard "query-response" kind of user interaction. All efforts directed at the user interface emphasize that the apparent intelligence of any knowledge-based system critically depends on the flexibility and richness of user-system interaction, and that the subtleties of descriptive responses and anticipatory replies are most important.

1.3.5. Logic databases

Here the predominant influence on DLOG comes from logic database research (e.g., [Gallaire78, Gallaire81]). The current logic database research relies on theorem-proving and logic programming developments of the seventies (e.g., [Kowalski79]), but owes a debt to earlier work whose foresight is often ignored, or taken for granted (e.g., [Levien67, Thompson69]). Much of the motivation for a logical treatment of databases can be credited to the efforts in applying theorem-proving techniques to relational data bases (e.g., [Minker75, Reiter78a]).

The most direct influences on DLOG come from Reiter [Reiter83] and Kowalski [Kowalski81]. These papers provided the theoretical background for the specification and implementation of a logic-based data model, and for dealing with the logic of data dictionaries and integrity constraints.

Although the motivation for using embedded descriptive terms comes from Bobrow and Winograd [Bobrow77b], similar motivation is evident in some logic database work (e.g., [Dilger78, Janas81]). In particular, the use of definite descriptions within queries [Dilger78] has motivated the investigation of DLOG's more elaborate descriptions.

1.4. Overview

Chapter 2 discusses the relationship between theories, interpretations, and models, and attempts to provide the groundwork for combining the concepts of a logic-based representation scheme and a data model. The distinction between databases viewed as interpretations or as models is discussed, and various conceptions of data models are examined before choosing one that correlates well with the notion of a logic-based representation scheme.

The use of logic as the foundation of a logic-based data model is discussed, and some results about the equivalence of logic databases and relational databases are reviewed.

The chapter concludes with an overview of the DLOG data model and a description of an example application. An implementation is sketched, and its use in developing a department database (DDB) is described. The DDB describes The University of British Columbia's undergraduate degree program in Computer Science. An example DDB terminal session demonstrates many of the DLOG system's facilities.

Chapters 3 and 4 provide a detailed description of the DLOG data model. Chapter 3 reiterates the data model concept adapted from various DBM data model definitions, and describes the four major components of DLOG: (1) the representation (data description) language, (2) the language semantics, (3) the proof procedure extensions for the language, and (4) the relationship between DLOG assertions, queries, and application databases.

Chapter 4 provides further details on the interpretation of DLOG terms. These include examples of various kinds of descriptive terms like lambda expressions, and definite and indefinite descriptions of individuals and sets. Some problems associated with descriptions are discussed (e.g., attributive versus referential use), and the DLOG interpretation for each kind of descriptive term is specified. This chapter ends with examples of how excerpts from the UBC Undergraduate Calendar can be represented by DLOG sentences.

Chapter 5 presents a brief overview of Prolog, and describes how the DLOG data model's specification can be converted into an implementation. Prolog's suitability as an implementation language is discussed, and the structure of a DLOG prototype is described. This description gives the details of each implementation component, and explains the interpretation of embedded terms via extended unification. The extended matching procedure includes a partial proof heuristic as a mechanism for evaluating queries whose deductive interpretation fails.

Chapter 6 explains how a Prolog-independent semantics of DLOG might be based on Montague's second order intensional logic. It is shown how DLOG can be expressed in terms of a second order language, and how Montague's second order intensional semantics must be extended to deal with individual descriptions.

The final chapter, chapter 7, concludes with an evaluation of DLOG's contributions to AI and DBM.

Chapter 2
Logic databases

2.1. Logic and Database Management perspectives

The introduction of §1.1.1 presented some recent views on the distinction between databases and knowledge bases. Those noted were admittedly weak, since they might evaporate when interpreted under suitable biases. If we equate the concepts of knowledge base and logic database, a much more incisive distinction is available: all knowledge bases can be viewed as logic databases because nearly all knowledge bases use some form of denotational semantics. Under this assumption, we can distinguish between databases as interpretation (i.e., traditional databases)[5] and databases as theories (i.e., knowledge databases or logic databases) [Nicolas78a, Reiter83, Kowalski81].

Briefly, the distinction is based on how information in a data base is perceived: a database consists of a conceptual schema and a collection of facts; a logic database is a uniform collection of asserted axioms with no explicit distinction between conceptual schema information and particular facts. The facts of a traditional database form an *interpretation* of the conceptual schema, while a logic database is a logical *theory* for which multiple interpretations may exist.

2.1.1. Theories, interpretations and models

A logical theory is a collection of sentences of a logical language, implicitly including all those facts deducible from it by the logic's proof procedure. Correspondences between sentences derivable in the theory and facts in a particular domain depend on the logic's properties of soundness and completeness.

[5] By "traditional database" we mean relational database. Because of the predominance of the relational data model in Database Management (DBM) theory, we adopt that model as representative. Subsequent uses of "DBM" refer to this representative data model.

By interpreting a database as a logical theory, we can use the logic's proof procedure to derive inferences *independent* from any style of observation or method of evaluation particular to the domain being represented. More simply, a proof procedure can answer questions that would otherwise require appeal to facts in the domain — logical consequence substitutes for truth.

Logical theories are related to real domains by using denotational semantics. This semantic theory specifies how to put the syntactic objects of a theory (i.e., the variables, constants, and predicates) in correspondence with the real objects of the domain (i.e., individuals and sets of individuals). Each set of correspondences is called an *interpretation* of that theory.

Interpretations are the mechanism by which we understand theories: we form an interpretation when conceiving axioms to include in a logical theory, and we use an interpretation to understand inferences drawn by a proof procedure. Each consistent (i.e., non-contradictory) theory has a subset of interpretations called models. A model is an interpretation that makes every sentence of the theory true.

To understand the difference between traditional and logic database approaches, consider two databases for a single domain, one viewed as a logical theory, the other as a model (i.e., an interpretation that makes each sentence of the theory true). A query to this pair can be answered in two ways: as an *alleged theorem* of the theory — successful derivation and recovery of variable bindings provides an answer; or, we can interrogate the model to see if the query is *true* — we evaluate the query in the model. The difference is the same as that exploited by Gerlenter in his geometry theorem proving program [Gerlenter63], and further elaborated by Reiter [Reiter76]. When answering a query, we can appeal to our knowledge (i.e., axioms) and inferencing mechanisms, or we can search for an answer directly, in the model.

2.1.2. Models versus theories

In contrast to viewing a database as a logical theory, the DBM view most often considers the tuple base as a model of the conceptual schema. In a sense, the DBM view captures the domain *inside* the machine, while the logical view maintains a more or less accurate representation or theory *about* the domain.

One important consequence of the DBM view is that the representation of incomplete information becomes difficult [Reiter83, Kowalski81]. With the model (i.e., tuple base) at its disposal, a DBM system can hardly represent a fact like "John or Joe loves Mary, but I don't know which," because one or the other must be the case. Similar difficulties exist with the treatment of existing but unknown individuals. As Reiter observes [Reiter83], handling of incomplete information from this view necessitates often complex model-theoretic manoeuvres, like the splitting of interpretations (e.g., [Lipski79]), or inclusion of special constants denoting various kinds of incompletely specified individuals (e.g., [Codd79, Vassiliou79]).

As will become clear, viewing a database as a logical theory offers several advantages, including uniformity of representation (i.e., no distinction between conceptual schema and tuple base) and uniformity of processing (proof procedure provides the mechanism for answering queries, enforcing constraints, and various other database operations).

2.2. Logic as a data model

The concept of a *data model* has been developed as a means of meeting the primary objective of DBM: data independence [Fry76]. The growing field of logic databases is founded on the analysis and reconstruction of DBM techniques, originally to augment DBM systems with inferential capabilities (e.g., [Minker75, Minker78, Reiter78a]), but more recently to replace them both as theoretical and practical models of managing information in a *data independent* fashion (e.g., [Kowalski78, Reiter83, Kowalski81]).

Logic database researchers have been astute in observing the advantages of using logic for data description, but the data model concept has not been generally acknowledged as significant. One exception is Colombetti et al., who are careful to point out that a data model is more than a language, and more than a data structure [Colombetti78]. Their minimal definition of a data model includes:

(1) a formal specification of all operations on objects belonging to the data model, and

(2) a formal specification of the semantics of the operations.

Though Colombetti et al. focus their attention on methods for algebraic specification of data model operations, their general concerns complement those of some Artificial Intelligence (AI) researchers who argue in favour of logical representation languages. For example, Hayes [Hayes74] proposes the concept of a *representation scheme,* which includes the language, semantics, and reasoning strategy of a knowledge representation system. Hayes' proposal is an attempt to provide a concept whereby disparate representation languages can be compared and evaluated. In fact a representation scheme is a kind of data model; its definition reflects the spirit of Codd's definition:[6]

(1) a collection of data structure types,

(2) a collection of operators or inferencing rules, and

(3) a collection of integrity rules.

The logical approach to databases combines Codd's first and third components into one: general axioms (a superset of integrity constraints) and ground axioms are sentences of the same language [Kowalski78].

The advantages of specifying the language and data operations are clear:

(1) there is no debate whether a data object is part of the model, and

(2) operations on data model objects are fully prescribed, precluding arbitrary operations that may render objects uninterpretable (cf. LISP-based representations like MICRO-PLANNER [Sussman71]).

The only discrepancy in consolidating the logic database, DBM and AI approaches is the question of including a semantics within the data model concept. The question of semantics is not addressed in Codd's formulation because of the DBM view of a database as an interpretation. The meaning of any

[6] [Codd81, p. 112].

particular collection of tuples is specified with general rules of the conceptual schema, not by any inherent semantics of the data model. The difficulty of incorporating more meaning into DBM data models has been attributed to the implicit physical record or tuple-oriented semantics (e.g., [Abrial74, Kent79]).

A logic-based data model requires a semantics because of the database as theory view. The semantics is a guide to the conception of axioms that describe a domain (e.g., what individuals correspond to constants?), and to the subsequent interpretation of inferences drawn by the data model's reasoning mechanism.

Yet another characterization of the data model concept is given by McLeod:[7]

(1) a data space: a collection of elements and relationships among the elements,

(2) a collection of type definition constraints to be imposed on the data space,

(3) manipulation operators supporting the creation, deletion and modification of elements, and

(4) a predicate language used to identify and select elements from the database.

Again, the logic database approach consolidates McLeod's first and second components by providing a language for asserting both specific and general axioms. McLeod's fourth component captures the notion of query language, and begs the issue of whether a query language should be included in a data model. The logical view works in either case, since a query can be viewed as an alleged theorem whose successful derivation results in an answer [Reiter78a, Minker78]. Alternatively, more sophisticated query facilities can be based on logic [Pirotte78, Emden78, Williams83].

Though inclusion of the query language within the data model might be debated,[8] the role of McLeod's third component is not clear. If "creation, deletion and modification" means manipulation of data model objects *as directed by the inferencing mechanism* (cf. Codd's second requirement), then the logic approach suffices.

[7] [McLeod81, p. 27].

[8] E.g., see [Codd81, p. 112].

The organization of logic databases and their maintenance has been investigated by several researchers [Colombetti78, Bowen82, Kowalski81]. Bowen and Kowalski's approach is the most interesting since it demonstrates that many aspects of logic database maintenance can be specified with logic. Logic can be used to describe the application domain, and the relationship between new assertions, queries, and the database can also be specified. *The same logic can be used for both purposes,* by combining the object level language used to formulate axioms of the domain, and the meta level language used to formulate the relationships between the axioms database, queries, and new axioms [Bowen82].

The reasoning component of the DLOG data model is specified in a way similar to that proposed by Bowen and Kowalski. The specification of the model's reasoning component will include the proof procedure of the object language and a specification of the relationship between application databases, queries, and assertions.

To summarize, a logic-based data model should include the following components:

(1) a specification of the syntax of the language used to define logic databases (and, possibly, queries),

(2) a specification of the way that sentences of the language are given meaning.

(3) a specification of the proof procedure or reasoning mechanism that is used to derive implicit relationships among database axioms,

(4) a specification of the relationship between application databases, queries, and assertions, i.e., a specification of the maintenance of application databases, independent of physical data considerations.

The abstract specification of such a data model can provide the basis for the analysis of the many competing data models [Reiter83]. In addition, while the systematic analysis of DBM capabilities in logical terms argues well for the use of logic for data description, the specification of a logic-based data model is a prerequisite for exploiting those advantages in an implementation.

2.2.1. What kind of logic?

Many researchers in logic databases use variants of first order logic tailored for their particular purpose (e.g., Horn clauses [Kowalski78], first order logic with equality [Reiter83], many-sorted first order logic [Minker78]). The strength of their conviction ranges from claims that it is the only reasonable choice [Kowalski79], to its justification on grounds of generalizability and uniformity [Reiter83].

As a mechanism for the analysis of data modelling techniques, first order logic has no equal — it is a *de facto* standard, much in the same way as the relational algebra is a standard for analyzing DBM data manipulation languages [Codd72].

2.2.2. Syntax, semantics and proof theory

In general, a reference to "logic" includes the components of a logical calculus (i.e., an alphabet, a syntax for formulas, logical axioms and a set of inference rules) and an appropriate semantic theory (e.g., Tarskian semantics).

The first order restriction is a *syntactic constraint* on the well-formedness of formulas. As noted above, the semantic theory prescribes how syntactic objects can be aligned with domain objects to determine the truth of formulas.

Though the notion of a "proof theory" has a clear definition in the logic literature (e.g., [Rogers71]), in logic database research the concept sometimes includes a particular strategy for its application. This accounts for the observation that, while first order proof theory is complete (i.e., all possible true formulas are derivable), the strategy for searching for proofs may be incomplete. For example, Reiter's use of "proof theory" [Reiter83] refers to first order proof theory exclusive of application strategy, while Kowalski's deduction mechanism [Kowalski78, Kowalski81] employs an efficient but incomplete depth-first strategy.

In AI, the appropriateness of logical syntax as a representation language has been hotly debated (e.g., see [Brachman80]). First order syntax is generally the least pleasing aspect of a user interface, but it can be appropriately "sugared." Indeed many purportedly non-logical languages can be re-expressed in a first order formulation [Hayes77, Hayes80].

Tarskian semantics is an appropriate semantic theory "off the shelf." In contrast to AI, where representation semantics are often specified by implementation, and DBM, where "semantic" data models are just becoming a concern, Tarskian semantics is a well understood method of ascribing meaning. A major advantage of a logic-based data model is the ability to analyze the equivalence of disparate representations by first rendering them in first order syntax, and then interpreting their meaning via Tarskian semantics [Hayes80].

Similarly, logical proof theory provides a standard of comparison for deductive reasoning mechanisms. Reasoning procedures based on first order proof theory originated with theorem-proving programs (e.g., [Gilmore60]), and have provided the foundation for extending existing DBM systems with deductive capabilities [Minker75, Kellogg78, Reiter78a].

2.2.3. Assumptions in logic-based data models

Though inferencing algorithms are an integral part of logic databases, adapting them for inclusion in a data model requires some special care. To illustrate, consider what is necessary to axiomatize the relational data model in first order logic. Recall from §2.1 that much of the difficulty in extending the DBM view of a database arises from the database as interpretation view. Query mechanisms often exploit several basic assumptions about a relational database to provide answers (e.g., [Reiter78b, Clark78, Reiter83]).

The kinds of assumptions made can be classified into two categories:

(1) assumptions about individuals, and

(2) assumptions about sets of individuals (i.e., predicate extensions).

In both cases the assumptions are used to interpret incomplete information. For example, under the DBM interpretation the query "Is there an Air Canada flight to Eagle's Crotch?" might be answered "No" on the assumption that the database is the unique model of the Air Canada flight information. Since no such flight is recorded, it must not exist. The negative answer is produced by failure to find an affirmative one.

In logic databases, the possibly many interpretations can be restricted by casting these assumptions as axioms of the theory. For the relation database interpretation, the following are necessary: Let $\alpha_1, \alpha_2, \cdots \alpha_n$ be a list of all individual constants in a database DB. Then

$$\forall x . x = \alpha_1 \vee x = \alpha_2 \vee \cdots \vee x = \alpha_n \qquad \text{(domain closure)}$$

is an axiom of DB. Notice that this is only possible under the assumption that the domains in question are all finite. Similarly, all assertions of the form

$$\alpha_i \neq \alpha_j \ \ \text{for} \, i \neq j \ \ 1 \leq i, j \leq n \qquad \text{(name uniqueness)}$$

are axioms of DB. For all n place predicates Φ_n of DB, let

$$\forall \vec{x}_n . \Phi_n(\vec{x}_n)$$

be an abbreviation for

$$\forall . x_1 x_2 \cdots x_n . \Phi_n(x_1, x_2, \cdots x_n)$$

and let

$$\Phi_n(\vec{c}_n)$$

be an abbreviation for

$$\Phi_n(c_1, c_2, \cdots c_n)$$

where c_i are individual constants of DB. The expression $\vec{c}_n$ denotes a tuple of n constants; let $T(\Phi_n) = \{\vec{c}_n \mid \Phi_n(\vec{c}_n) \epsilon \text{DB}\}$ be the set of all tuples such that $\vec{c}_n$ is in the extension of the relation Φ_n in DB. Now if $T(\Phi_n)$ is empty, then

$$\forall \vec{x}_n . \neg \Phi(\vec{x}_n) \qquad \text{(predicate completion)}$$

is an axiom of DB. Otherwise, for all $\vec{c}_{n_i} \epsilon T(\Phi_n)$, and m the cardinality of $T(\Phi_n)$,

$$\forall \vec{x}_n . \Phi_n(\vec{x}_n) \supset \vec{x}_n = \vec{c}_{n_1} \vee \vec{x}_n = \vec{c}_{n_2} \vee \cdots \vee \vec{x}_n = \vec{c}_{n_m} \qquad \text{(predicate completion)}$$

is an axiom of DB. The notation $\vec{x}_n = \vec{c}_{n_1}$ is an abbreviation for $x_1 = c_{1_1} \wedge x_2 = c_{2_1} \wedge \cdots x_n = c_{n_1}$.

The first two axioms state that the known individual constants are all and only those that exist, and furthermore, that they are unique. The predicate completion axioms assert that the known facts are all that exist — if some fact is not included, its negation may be inferred. Reiter [Reiter83] provides details, including modifications for various kinds of incomplete information, and theorems that demonstrate the equivalence of particular classes of relational models with their logic counterparts.

In general, these assumptions are cast as axioms to give a first order interpretation of their meaning. An implementation of a reasoning mechanism may forego the repeated application of these axioms in favour of a more efficient mechanism. For example, Clark [Clark78] shows how the negation-as-failure proof mechanism of Prolog can be interpreted as an efficient application of the if-and-only-if form of the predicate completion axioms. Because the axiomatic specification of these underlying assumptions is possible, a logic-based data model can include a clear indication of its assumptions regardless of implementation.

2.3. An example: a departmental database in DLOG

The motivation for DLOG's development is twofold: it is a knowledge representation language and a logic-based data model. The design of the *language* is motivated by an AI representation concern: the ease of mapping an application domain to a knowledge base. The design of the DLOG data model and interpreter has been influenced by a traditional DBM goal: to provide for the domain independent storage and manipulation of data.

The implemented DLOG system provides facilities for the storage and manipulation of information stored in accord with the DLOG data model. Its design has been guided by the DBM objective of data independence [Fry76].

The issue of data independence has not had great impact on AI representation systems, just as researchers in DBM have been reluctant to face the issue of making inferences from stored information. The predominant DBM view of information has been as physical data to which sophisticated manipulation mechanisms can be applied [Abrial74, Kent79]. In contrast, the DLOG data model views an application database as a logical theory: the information content of the data base includes not only the asserted information, but all those inferences that can be drawn by the data model's inferencing mechanism.

2.3.1. The example domain

The prototype DLOG implementation has been exercised in the domain of undergraduate degree programs at The University of British Columbia. This is *not* merely a domain from which examples were drawn to illustrate DLOG capabilities. The exercise includes a complete specification of the requirements of a Bachelor of Science with a Major in Computer Science. The knowledge represented is the factual information that is expected of a program advisor who counsels students in course selection and degree requirements. Note that this does not include the wealth of pragmatic and commonsense knowledge that a typical advisor may have acquired through experience. The Departmental Database (DDB) includes a simple facility for creating and maintaining student transcripts, together with descriptions of the degree program options for which students are eligible. The transcript maintenance component is included to exercise the DLOG DDB. It does not comprise a general solution to the difficult problem of maintaining a general description of both degree program and student evolution. The DDB is intended to be a description of *one* state of the degree program requirements, and no attempt has been made to develop a representation that will correctly represent and efficiently maintain such concepts as "changes to the degree program will not affect students whose registration date precedes those changes."

The prototype DLOG implementation described in chapter 5 does, however, support certain "STRIPS-like" database update operations to be embedded in DLOG applications. For example, in the DDB domain a STRIPS-like add/delete pair is used to change the course status of a student from "enrolled" to "completed."Further details about possible alternatives to this approach are briefly discussed in §§3.6, 5.3.2, 7.4.

From the AI viewpoint, the DLOG system and DDB is a kind of student advisor: with suitable modifications, this system might have been presented as an expert "advisor."

2.3.2. Departmental database overview

The basic structure of the DDB system is illustrated in fig. 2.1. The host operating system provides standard facilities like file management, and the DLOG system provides the facilities for the

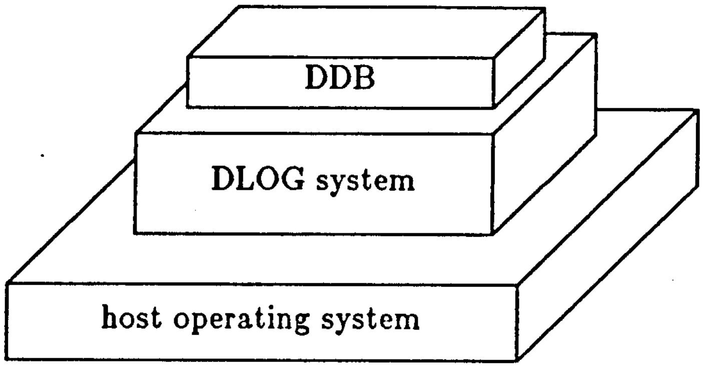


Figure 2.1 DDB system structure.

management of DLOG application databases. DLOG prototypes exist for both the Michigan Terminal System (MTS) and IBM's Conversational Monitor System (CMS).

The finer structure of the DLOG system reveals a Prolog interpreter within which DLOG is defined (fig. 2.2).

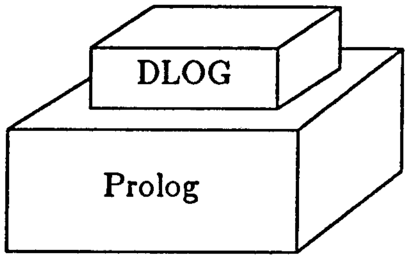


Figure 2.2 DLOG system structure

This style of construction is similar to the way MICRO-PLANNER [Sussman71] or KL-ONE [Brachman79] are constructed on top of LISP interpreters. An important difference is that *all* DLOG assertions are defined in the DLOG data model: its specification does not allow arbitrary use of the defining interpreter.[9] This means that *all* assertions within the data model specification are interpretable via the semantics of the DLOG data model — no appeal to the implemented system or its implementation language is necessary.

The complete DLOG implementation currently consists of 478 Prolog assertions, 355 of which are non-trivial implications. This total includes the DLOG proof procedure, parser, extended meta predicates, menu interfaces, and input/output predicates.

The DDB can be viewed as a monolithic collection of DLOG assertions, but it too has a finer structure that reveals the construction and use of its various components. One view separates the DDB into degree program knowledge and student transcripts, as in fig. 2.3.

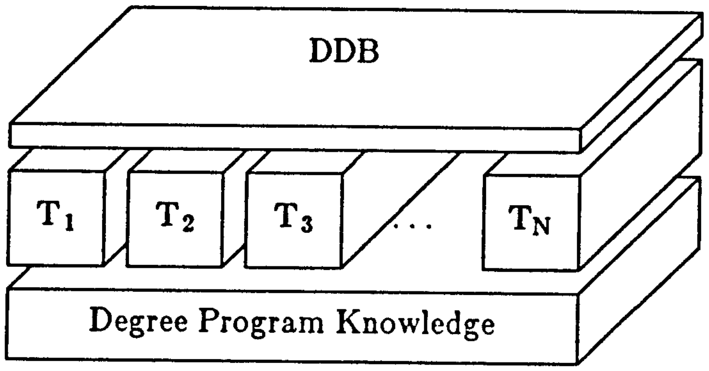


Figure 2.3 DDB knowledge: database designer view

The degree program knowledge includes assertions about program requirements, prerequisites and promotion, while student transcripts record facts pertinent to particular students.[10] This distinction between degree program knowledge and student transcripts is not part of DLOG. It is a result of the

[9] However, a restricted use is possible, in order to define application dependent software (see §5.3.1). The application programming is not provided with unrestricted access to the system's proof procedure.

[10] A preliminary understanding of the relationship between transcripts and degree program knowledge can be had by viewing a student transcript as an individual frame (AI readers), or as an external schema (DBM readers). These conceptualizations will aid understanding until details are provided.

structure inherent in the domain, and is the kind of classification conceived by the database designer or "knowledge engineer." In the real world, transcripts change more quickly than the degree programs that guide their construction. The distinction is local to the DDB database, independent of DLOG.

Another view of the DDB distinguishes three classes of assertions: question answering knowledge, integrity constraints and meta or data dictionary knowledge (see fig. 2.4).

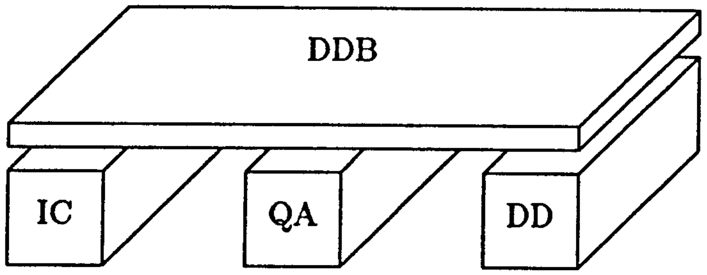


Figure 2.4 DDB Knowledge: DLOG view

The DLOG interpreter does recognize this classification, and uses each class appropriately.

The DDB application database currently consists of 667 DLOG assertions, only 74 of which are implications. The domain definitions (i.e., unary predications) and question-answering assertions number 524, and the rest are integrity constraints, data dictionary assertions, and topic dictionary assertions. (See Appendix 4 for a complete listing of the DDB.)

The DDB was designed to be helpful in constructing a consistent description of degree program requirements and constraints, and to be helpful in creating and maintaining transcripts of students pursuing various programs of study. During specification, inconsistencies in the actual calendar description were discovered as a result of attempting to describe it in DLOG.

The example terminal session (fig. 2.5) demonstrates some of the capabilities of the existing system. The terminal session below used about 28 seconds of CPU time on a lightly loaded IBM 4341 processor, running IBM's Conversational Monitor System (CMS) under the Virtual Machine (VM) resource manager. Waterloo IBM Prolog uses 25K bytes of mainstore. The DLOG implementation uses another 50K bytes, and the DDB 118K bytes more. The loaded DDB system occupies 193K bytes of mainstore. The space utilization of DLOG is difficult to estimate because the Waterloo IBM Prolog interpreter does not do any compaction after assertions have been deleted. The session of fig. 2.5 required one reloading in order to provide the space necessary for the DLOG implementation to perform the required computation of set extensions.

This session transcript is intended to arouse the reader's curiosity so he will continue reading to uncover the details of how some function or feature is provided. The Prolog source code for the DLOG implementation used here is in Appendix 1.

A detailed descriptions of each DDB predicate is given in Appendix 2. In the session below, a brief summary of each such description appears immediately after the first use of DDB predicate. In addition, the session is sprinkled with comments that are intended to help the reader interpret the output syntax of the Waterloo Prolog interpreter.

```
R;
ddb                                        This CMS command file loads Waterloo PROLOG, builds DLOG,
WELCOME TO WATERLOO PROLOG 1.1             and loads the DDB application database...
LOAD(OPS1)<-
LOAD(ALL1)<-
LOAD(ASSERT1)<-
LOAD(BROWSE1)<-
LOAD(CTX1)<-
LOAD(DERIVE1)<-
LOAD(EXT1)<-
$$$$$$$$
LOAD(IC1)<-
LOAD(INPUT1)<-
LOAD(META1)<-
LOAD(PARSE1)<-
LOAD(PREDS1)<-
LOAD(QUERY1)<-
LOAD(SETFNS1)<-
$$$$$$$$
LOAD(SORT1)<-
LOAD(START1)<-
LOAD(SYNTAX1)<-
LOAD(TRANS1)<-
LOAD(UNIFY1)<-
$$$$$$$$
LOAD(UTILS1)<-                             This ends the definition of DLOG, now the various DDB
LOAD(DDBQA1)<-                             components are loaded...
$$$$$$$$
LOAD(DDBQA2)<-
```

```
$$$$$$$$
LOAD(DDBQA3)<-
LOAD(DDBDOM)<-
LOAD(DDBIC)<-
LOAD(DDBDD)<-
$$$$$$$$
LOAD(DDBTOP)<-
LOAD(DDBCOM)<-
LOAD(DDBTDIR)<-
DLOG 1.0

browse.                                       The browse command prompts with a menu of alternatives...
1: Topics
2: Constraints
3: User predicates
4: System predicates
5: Enter predicate
6: Enter skeleton

 Selection? 1                                 This selection prompts for a topic, and uses the topic dictionary
 Topic? admission                             to find relevant predicate names.
Predicates relevant to admission
dept_program_prereq
faculty_program_prereq
program_prereq
```

dept_program_prereq(u,v,w,x,y,z). This predicate asserts that course z is a requirement in year y, for any degree program offered in department u, at level v (e.g., Bachelor, Master, etc.), in stream w (e.g., majors, honours, etc.), with field x (e.g., Computer Science, Physics-Mathematics, etc.). Requirements stated in this way provide the details of courses specified at the department level, and can be inherited by program course requirements.

faculty_program_prereq(v,w,x,y,z). This predicate is used to specify faculty level requirements z that are prerequisite to enrolling in year y of a degree program with stream x, at level w in faculty v. As for *dept_program_prereq*, these prerequisites are a subset of those specified by *program_prereq* for the appropriate degree programs.

program_prereq(x,y,z). This predicate asserts that requirement z is a prerequisite for enrolment in year y of program x.

```
Another topic?                                Many of the menus prompt for continue or exit...
 Acknowledge(y|n): n
Continue browsing?
 Acknowledge(y|n): y

1: Topics
2: Constraints
3: User predicates
4: System predicates
5: Enter predicate
6: Enter skeleton

Selection? 5                                  Having identified a predicate name of interest, the user requests
Predicate? program                            the retrieval of a few appropriate assertions...
program(BScMajorsCS)
program(BScHonoursCS)
program(BScHonoursCSPHYS)
program(BScHonoursCSMATH)
```

program(x). This predicate specifies the domain of known degree programs (e.g., BScMajorsCS, BScHonoursCS, etc.).

```
Continue browsing?
 Acknowledge(y|n): y

1: Topics
2: Constraints
3: User predicates
4: System predicates
5: Enter predicate
6: Enter skeleton

 Selection? 5
 Predicate? program_prereq
program_prereq(*1,first,lambda(*2,age_of(*2,*3)&GE(*3,16)))
program_prereq(*1,*2,*3)<-
    faculty_of(*1,*4)
   &level_of(*1,*5)
   &stream_of(*1,*6)
   &faculty_program_prereq(*4,*5,*6,*2,*3)
program_prereq(*1,*2,*3)<-
    dept_of(*1,*4)
   &level_of(*1,*5)
   &stream_of(*1,*6)
   &field_of(*1,*7)
   &dept_program_prereq(*4,*5,*6,*7,*2,*3)
Continue browsing?
 Acknowledge(y|n): y
```

The first few axioms for the *program_prereq* predicate are listed...notice that this is retrieval or metalevel inference: each assertion retrieved is the result of a demonstrating that there exists an axiom of the current database that has 'program_prereq' as the predicate of the consequent.
The output form ' C <- A1 & A2 & ... & An' is the Waterloo Prolog syntax for the implication 'A1 & A2 & ... & An -> C.'
The notation '*n', where n=1,2,3..., denotes a variable.

```
1: Topics
2: Constraints
3: User predicates
4: System predicates
5: Enter predicate
6: Enter skeleton

 Selection? 6
 Skeleton (end with .)? program_prereq(BScMajorsCS,first,*).
program_prereq(BScMajorsCS,first,lambda(*1,age_of(*1,*2)&GE(*2,16)))
Browse implications?
 Acknowledge(y|n): y
program_prereq(BScMajorsCS,first,*1)<-
    faculty_of(BScMajorsCS,*1)
   &level_of(BScMajorsCS,*2)
   &stream_of(BScMajorsCS,*3)
   &faculty_program_prereq(*1,*2,*3,first,*4)
program_prereq(BScMajorsCS,first,*1)<-
    dept_of(BScMajorsCS,*1)
   &level_of(BScMajorsCS,*2)
   &stream_of(BScMajorsCS,*3)
   &field_of(BScMajorsCS,*4)
   &dept_program_prereq(*1,*2,*3,*4,first,*5)
More?
 Acknowledge(y|n): n
Continue browsing?
 Acknowledge(y|n): n
```

This selection will read a term, and use it as a pattern to retrieve the relevant predicates...
This atomic clause asserts that you must be at least 16 to register in the BScMajors program.
It is represented as a program prerequisite for first year.
The lambda expression form permits the assertion of a relation on predicates.

These two implications assert that the faculty and department requirements for the BScMajors program are merely specializations of program prerequisites.

```
<-set(*x:program_prereq(BScMajorsCS,first,*x)).
Query: set(*1:program_prereq(BScMajorsCS,first,*1))
Result: set(lambda(*1,completed(*1,an(*2,topic_of(*2,Science)
              &(course_no(*2,11)|course_no(*2,12))))
              |permission(*1,*3,Science)&dean(*3,Science))
          &lambda(*4,completed(*4,PHYS11))
          &lambda(*5,completed(*5,ALGEBRA12))
          &lambda(*6,completed(*6,ALGEBRA11))
          &lambda(*7,completed(*7,CHEM11))
          &lambda(*8,age_of(*8,*9)&GE(*9,16)))
```

This is the first DLOG query...it requests the set of prerequisites for year one of the BScMajorsCS program...
...each prerequisite is a lambda term representing a predicate that must be satisfied.

This says that you must have completed a Science course numbered 11 or 12 (or have the dean's permission), have completed the other matriculation courses listed, and be at least sixten years old.

```
transcript.
```

This is the first invocation of the application-dependent command that helps the user maintain student transcripts.

```
1: load
2: save
3: list
4: edit
5: create
6: browse
```

This option knows something about the information needed to construct a new transcript...

```
 Selection? 5
 New name identifier? Smith
 Program name? BScMajrosCS
Unknown program...retry?
 Acknowledge(y|n): y
 Program name? BScMajorsCS
```

After identifying the prospective student, and program, the transcript feature uses the DLOG transaction processor to accumulate enrolment data...

```
Enter admissions data (Type end to stop):
Begin transaction
completed(Smith,ALGEBRA11).
Assert atom: completed(Smith,ALGEBRA11)
completed(Smith,ALGEBRA12).
Assert atom: completed(Smith,ALGEBRA12)
completed(Smith,CHEM11).
Assert atom: completed(Smith,CHEM11)
completed(Smith,CHEM12).
Assert atom: completed(Smith,CHEM12)
completed(Smith,PHYS11).
Assert atom: completed(Smith,PHYS11)
```

Each assertion is acknowledged, and a DLOG transaction command produces a list of active assertions...

```
list.
1: completed(Smith,ALGEBRA11)
2: completed(Smith,ALGEBRA12)
3: completed(Smith,CHEM11)
4: completed(Smith,CHEM12)
5: completed(Smith,PHYS11)
```

...the "end" command stops transaction input and invokes the integrity processor...

```
end.
Update DB with transaction?
 Acknowledge(y|n): y
```

The DLOG transaction processor reports completion, then the transcript processor takes over...

```
Transaction completed
Failed to satisfy: age_of(Smith,*1)&GE(*1,16)
Smith not eligible for first year BScMajorsCS
```

...satisfaction of the prerequisites will sanction transcript creation...

```
Augment admissions data?
 Acknowledge(y|n): y
Continue transcript creation...
```

DLOG's transaction processor is used again...

```
Enter admissions data (Type end to stop):
Begin transaction

age_of(Smith,17).
Assert atom: age_of(Smith,17)
end.
Update DB with transaction?
 Acknowledge(y|n): y
Transaction completed
```

The transcript command acknowledges creation of the transcript...

```
Assert atom: name_id(Smith)
Smith transcript created
Continue?
 Acknowledge(y|n): y

1: load
2: save
3: list
4: edit
5: create
6: browse

 Selection? 3
 Student identifier? Smith
```

...and will produce a listing of the transcript.

```
Transcript of Smith
age_of(Smith,17).
```

```
completed(Smith,ALGEBRA11).
completed(Smith,ALGEBRA12).
completed(Smith,CHEM11).
completed(Smith,CHEM12).
completed(Smith,PHYS11).
Continue?
 Acknowledge(y|n): n


<-eligible_for_admission(Smith,BScMajorsCS).             The user queries to verify Smith's eligibility for
Query: eligible_for_admission(Smith,BScMajorsCS)         admission into the BScMajorsCS program...
$$$$$$$$
Succeeds: eligible_for_admission(Smith,BScMajorsCS)


browse.                                                  ...and begins to investigate the courses required.

1: Topics
2: Constraints
3: User predicates
4: System predicates
5: Enter predicate
6: Enter skeleton

 Selection? 1
 Topic? courses
Predicates relevant to courses
faculty_elective
elective
unit_value
course_equivalent
course_prereq
eligible_for_course
student_program_contribution
program_contribution
dept_program_req
faculty_program_req
program_req
```

faculty_elective(x,y,z). This predicate asserts that course z can be considered to be a course from faculty y, when considered as an elective for programs offered in the faculty x.

elective(x,y). This predicate asserts that course y is a legal elective for program y. It is a weaker assertion than *faculty_elective*(x,y,z), since it does not specify which faculty the course is an elective for, nor what faculty the course is presumed to be from (see below).

unit_value(x,y). This predicate asserts that course x has unit value y.

course_equivalent(x,y). This predicate asserts that course x is viewed as equivalent to course y. This is used for cross listed courses, or for those which are similar enough so that credit cannot be had for both.

course_prereq(x,y). This predicate is used to assert that course x requires the satisfaction of requirements y. In this axiomatization of the application domain, these requirements are specified as lambda expressions to be satisfied (see §4.1.2).

eligible_for_course(x,y). This predicate is true when student x has satisfied the prerequisites for course y.

student_program_contribution(x,y,z). The course z will fulfill some requirement toward the completion of program y for student x. This predicate is derivable when the course z has not yet been taken by student x, but if completed, would make a contribution to the completion of program y.

program_contribution(x,y,z). This predicate is true when course z will make a contribution toward the requirements of completing year y of the degree program in which student x is enrolled. The derivation of this predicate requires the use of the DLOG predicate *extends* (see §5.4).

program_req(x,y,z). This predicate asserts that z is a requirement of year y for degree program x.

faculty_program_req(v,w,x,y,z). This predicate is used to specify general program requirements set at the faculty level (e.g., a certain number of Science units). It asserts that requirement z must be satisfied for year y of all programs with stream x, level w, and faculty v.

program_req(x,y,z). This predicate asserts that z is a requirement of year y for degree program x.

```
Another topic?
 Acknowledge(y|n): n
Continue browsing?
 Acknowledge(y|n): n
```

Many predicates are relevant to "courses," so the course requirements are sought in a more direct way...the query asks for "the set of all requirements for the first year of the BScMajorsCS program..."

```
<-set(*x:course_req(BScMajorsCS,first,*x)).

Query: set(*1:course_req(BScMajorsCS,first,*1))
Result: set(lambda(*1,completed(*1,ENGL100))
      &lambda(*2,completed(*2,CHEM110)!completed(*2,CHEM120))
      &lambda(*3,completed(*3,PHYS110)!completed(*3,PHYS115)
             !completed(*3,PHYS120))
      &lambda(*4,completed(*4,MATH100)&completed(*4,MATH101)
             !completed(*4,MATH120)&completed(*4,MATH121))
      &lambda(*5,completed(*5,CS115)!completed(*5,CS118)
             &completed(*5,an(*6,elective(BScMajorsCS,*6)
             &unit_value(*6,15)))))
```

```
<-eligible_for_course(Smith,CS115).
Query: eligible_for_course(Smith,CS115)
Failed to satisfy: (completed(Smith,*1)|course_enrolled(Smith,*1))
   &course_equivalent(*1,MATH100)&~completed(Smith,CS118)
Not deducible
```

Smith is not eligible for CS115 because he has not satisfied the necessary prerequisites...

```
<-course_prereq(CS115,*x).
Query: course_prereq(CS115,*1)
Succeeds: course_prereq(CS115,lambda(*1,(completed(*1,*2)
                              |course_enrolled(*1,*2))
                              &course_equivalent(*2,MATH100)
                              &~completed(*1,CS118)))
```

...therefore the prerequisites are sought.

The DLOG command processor is stopped, to enable heuristic mode (see §5.4). Heuristic mode is not always in effect in this experimental interpreter because it causes extensive backtracking and is therefore expensive...

```
stop.
EXIT DLOG
start<-
heuristic_mode.
<-restart.
DLOG 1.0
```

...however it does permit some queries (which would normally fail) to complete, subject to certain assumptions.
An English reading might be "Must one satisfy the requirement of completing CS115 for the course requirements of the first year of the BScMajorsCS program?"
The *extends(x,y)* predicate determines if a proof of the sentence *x* is a subproof of the sentence *y*,

```
<-course_req(BScMajorsCS,first,lambda(*x,completed(*x,CS115))).
Query: course_req(BScMajorsCS,first,lambda(*1,completed(*1,CS115)))
Heuristic assumption:
 completed(*1,CS115)
 extends
 completed(*1,CS115)!completed(*1,CS118)
&completed(*1,an(*2,elective(BScMajorsCS,*2)&unit_value(*2,15)))
Succeeds: course_req(BScMajorsCS,first,lambda(*1,completed(*1,CS115)))
```

```
course_enrolled(Smith,MATH100).
Assert atom: course_enrolled(Smith,MATH100)
```

Now the user begins enrolling Smith in the required course...

```
course_enrolled(Smith,MATH101).
Assert atom: course_enrolled(Smith,MATH101)


course_enrolled(Smith,PHYS120).
$$$$$$$$
Assert atom: course_enrolled(Smith,PHYS120)


course_enrolled(Smith,CHEM120).
Assert atom: course_enrolled(Smith,CHEM120)


course_enrolled(Smith,ENGL100).
Assert atom: course_enrolled(Smith,ENGL100)


transcript.
```

A quick check of the transcript will review Smith's status...

```
1: load
2: save
3: list
4: edit
5: create
6: browse

Selection? 3
Student identifier? Smith
Transcript of Smith
age_of(Smith,17).
completed(Smith,ALGEBRA11).
completed(Smith,ALGEBRA12).
completed(Smith,CHEM11).
completed(Smith,CHEM12).
completed(Smith,PHYS11).
course_enrolled(Smith,MATH100).
course_enrolled(Smith,MATH101).
course_enrolled(Smith,PHYS120).
course_enrolled(Smith,CHEM120).
course_enrolled(Smith,ENGL100).
Continue?
Acknowledge(y|n): n
```

Having enrolled in MATH100 and MATH101, he should be able to enroll in CS115...

```
course_enrolled(Smith,CS115).
Assert atom: course_enrolled(Smith,CS115)
```

...and complete registration with the appropriate assertion.
Integrity constraints on the "registered" predicate ensure that the student is eligible...

```
registered(Smith,BScMajorsCS,first).
$$$$$$$
Assert atom: registered(Smith,BScMajorsCS,first)


transcript.

1: load
2: save
3: list
4: edit
5: create
6: browse

Selection? 3
Student identifier? Smith
Transcript of Smith
age_of(Smith,17).
registered(Smith,BScMajorsCS,first).
completed(Smith,ALGEBRA11).
completed(Smith,ALGEBRA12).
```

...the updated transcript confirms the registration.

```
completed(Smith,CHEM11).
completed(Smith,CHEM12).
completed(Smith,PHYS11).
course_enrolled(Smith,MATH100).
course_enrolled(Smith,MATH101).
course_enrolled(Smith,PHYS120).
course_enrolled(Smith,CHEM120).
course_enrolled(Smith,ENGL100).
course_enrolled(Smith,CS115).
Continue?
 Acknowledge(y|n): y

1: load
2: save
3: list
4: edit
5: create
6: browse

 Selection? 2
 Student identifier? Smith
Smith transcript saved
Continue?
 Acknowledge(y|n): n
```

Another function of the transcript command is to save the transcript of the specified student...

```
head(PCGilmore,CS).
Failed constraint: faculty_member(PCGilmore)
Constraints failed for head(PCGilmore,CS)
```

A new assertion demonstrates the need of the DLOG transaction processor...

the assertion failed because of an integrity constraint that requires heads to be faculty members. Constraints are not used for deduction, so the assertion fails (see §3.2.3).

```
transaction.
Begin transaction

head(PCGilmore, CS).
Assert atom: head(PCGilmore,CS)
faculty_member(PCGilmore).
Assert atom: faculty_member(PCGilmore)
list.
1: head(PCGilmore,CS)
2: faculty_member(PCGilmore)
end.
Update DB with transaction?
 Acknowledge(y|n): y
Transaction completed
```

...the order of assertions within a transaction is irrelevant.

```
<-set(*x:head(*x,*y)).
Query: set(*1:head(*1,*2))
Result: set(PCGilmore)
```

...the set of all heads is a singleton.

Now the user will attempt to update Smith's grades...

```
<-completed(Smith,CS115).
Query: completed(Smith,CS115)
Not deducible
```

...first notice that Smith hasn't completed CS115...

```
completed(Smith,CS115).
Failed constraint: matriculation_course(CS115)
              |course_enrolled(Smith,CS115)
                &grade_of(Smith,CS115,*1)
                &passing_grade_of(CS115,*2)
                &GE(*1,*2)
                &DELAX(course_enrolled(Smith,CS115))
Constraints failed for completed(Smith,CS115)
```

...and that simply asserting so will not succeed.

```
grade_of(Smith,CS115,88).
Assert atom: grade_of(Smith,CS115,88)
```

A passing grade will help...

```
<-course_enrolled(Smith,CS115).
Query: course_enrolled(Smith,CS115)
Succeeds: course_enrolled(Smith,CS115)
```

(still enrolled...)

```
completed(Smith,CS115).
Assert atom: completed(Smith,CS115)
```

...and furthermore, a successful assertion of completion will invoke a transition constraint that retracts the old assertion that states that states Smith is enrolled...

```
<-course_enrolled(Smith,CS115).
Query: course_enrolled(Smith,CS115)
Not deducible .
```

...as verified by failure of this query.

```
<-completed(Smith,CS115).
Query: completed(Smith,CS115)
Succeeds: completed(Smith,CS115)
```

```
grade_of(Smith,CHEM120,77).
Assert atom: grade_of(Smith,CHEM120,77)
```

Now Smith's grades for the rest of his courses are entered...

```
grade_of(Smith,PHYS120,98).
Assert atom: grade_of(Smith,PHYS120,98)

grade_of(Smith,MATH100,87).
$$$$$$$$
Assert atom: grade_of(Smith,MATH100,87)

completed(Smith,CHEM120).
Assert atom: completed(Smith,CHEM120)
```

Assertions of both grade and completion are required, as no knowledge of completion requirements are encoded, e.g., attending labs, lectures, etc.

```
transcript.

1: load
2: save
3: list
4: edit
5: create
6: browse

Selection? 3
Student identifier? Smith
Transcript of Smith
age_of(Smith,17).
registered(Smith,BScMajorsCS,first).
completed(Smith,ALGEBRA11).
completed(Smith,ALGEBRA12).
completed(Smith,CHEM11).
completed(Smith,CHEM12).
completed(Smith,PHYS11).
completed(Smith,CS115).
completed(Smith,CHEM120).
grade_of(Smith,CS115,88).
grade_of(Smith,CHEM120,77).
grade_of(Smith,PHYS120,98).
grade_of(Smith,MATH100,87).
course_enrolled(Smith,MATH100).
course_enrolled(Smith,MATH101).
course_enrolled(Smith,PHYS120).
course_enrolled(Smith,ENGL100).
Continue?
 Acknowledge(y|n): n


completed(Smith,MATH100).
Assert atom: completed(Smith,MATH100)
```

...and the assertions about their completion

```
completed(Smith,PHYS120).
Assert atom: completed(Smith,PHYS120)


transcript.

1: load
2: save
3: list
4: edit
5: create
6: browse

 Selection? 3
 Student identifier? Smith
Transcript of Smith
age_of(Smith,17).
registered(Smith,BScMajorsCS,first).
completed(Smith,ALGEBRA11).
completed(Smith,ALGEBRA12).
completed(Smith,CHEM11).
completed(Smith,CHEM12).
completed(Smith,PHYS11).
completed(Smith,CS115).
completed(Smith,CHEM120).
completed(Smith,MATH100).
completed(Smith,PHYS120).
grade_of(Smith,CS115,88).
grade_of(Smith,CHEM120,77).
grade_of(Smith,PHYS120,98).
grade_of(Smith,MATH100,87).
course_enrolled(Smith,MATH101).
course_enrolled(Smith,ENGL100).
Continue?
 Acknowledge(y|n): n


grade_of(Smith,MATH101,86).
Assert atom: grade_of(Smith,MATH101,86)


completed(Smith,MATH101).
Assert atom: completed(Smith,MATH101)


grade_of(Smith,ENGL100,99).
Assert atom: grade_of(Smith,ENGL100,99)


completed(Smith,ENGL100).
Assert atom: completed(Smith,ENGL100)


transcript.

1: load
2: save
3: list
4: edit
5: create
```

```
6: browse

 Selection? 3
 Student identifier? Smith
Transcript of Smith
age_of(Smith,17).
registered(Smith,BScMajorsCS,first).
completed(Smith,ALGEBRA11).
completed(Smith,ALGEBRA12).
completed(Smith,CHEM11).
completed(Smith,CHEM12).
completed(Smith,PHYS11).
completed(Smith,CS115).
completed(Smith,CHEM120).
completed(Smith,MATH100).
completed(Smith,PHYS120).
completed(Smith,MATH101).
completed(Smith,ENGL100).
grade_of(Smith,CS115,88).
grade_of(Smith,CHEM120,77).
grade_of(Smith,PHYS120,98).
grade_of(Smith,MATH100,87).
grade_of(Smith,MATH101,86).
grade_of(Smith,ENGL100,99).
Continue?
 Acknowledge(y|n): n


browse.                                          Now the user wants information about the next year
                                                 of this program...
1: Topics
2: Constraints
3: User predicates
4: System predicates
5: Enter predicate
6: Enter skeleton

 Selection? 1
 Topic? year                                     ...this topic is not in the topic dictionary...
Unknow topic: year
Retry?
 Acknowledge(y|n): y
 Topic? eligible
Unknow topic: eligible
Retry?
 Acknowledge(y|n): y
 Topic? promotion
Predicates relevant to promotion                 ...but this one will help.
eligible_for_degree
eligible_for_year
registered
faculty_grad_req
grad_req
```

eligible_for_degree(x,y). This predicate is true when student x has satisfied all the graduation requirements for degree program y. Derivation of this relation as a query initiates the most complex and time consuming computation possible in the version of DLOG in which this application data base was developed.

eligible_for_year(x,y). This predicate is true when student x has completed the prerequisites for admission to year y of the program they are currently enrolled in.

registered(x,y,z). This asserts that student x is registered in year z of degree program y. This assertion cannot be made unless the student in question has satisfied the prerequisites for year z of program y.

faculty_grad_req(w,x,y,z). This predicate asserts that z is a graduation requirement for all degree programs from faculty w at level x (e.g., bachelor, master, etc.), in stream y (e.g., majors, honours, etc.). The set of faculty graduation requirements are a subset of the extension of the more general predicate *graduation_req*(x,y) for appropriate programs x.

grad_req(x,y). This predicate asserts that requirement y is one requirement to be satisfied in order to graduate with degree x.

```
Another topic?
 Acknowledge(y|n): n
Continue browsing?
 Acknowledge(y|n): y


1: Topics
2: Constraints
3: User predicates
4: System predicates
5: Enter predicate
6: Enter skeleton

 Selection? 5
 Predicate? eligible_for_year
eligible_for_year(*1,*2)<-                               Notice that testing for eligibility for a year involves
    program_enrolled(*1,*3)                              retrieving and testing all the previous year's requirements,
   &previous_year(*2,*4)                                 as well as retrieving and satisfying all prerequisites
   &satisfied (*1,set(*5:program_req(*3,*4,*5)))         of the next year.
   &satisfied(*1,set(*6:program_prereq(*3,*2,*6)))
Continue browsing?
 Acknowledge(y|n): n


<-eligible_for_year(Smith,second).                       The student has satisfied the appropriate requirements...
Query: eligible_for_year(Smith,second)
$$$$$$$$
$$$$$$$$
$$$$$$$$
$$$$$$$$
Succeeds: eligible_for_year(Smith,second)                ...and DLOG verifies this in approximately 8 cpu seconds.


stop.                                                    This stops the DLOG proof loop...
EXIT DLOG
START<-
<-STOP.                                                  ...and this stops PROLOG.
R;
```

Figure 2.5 An example DDB terminal session

Chapter 3
The DLOG data model

The DLOG data model is the foundation of the DLOG database management system. It consists of a representation language and corresponding semantics, a proof procedure, and a specification of the relationship between assertions, queries, and DLOG databases.

The description of DLOG is based on a 3-sorted first order syntax, standard Tarskian semantics, and a first order resolution proof theory augmented with various procedures for manipulating descriptions, lambda terms, and constraints. A later chapter (Ch. 6) proposes a non-standard approach to the formal semantics of the DLOG language.

The 3-sorted syntax is intended to reflect the intuitive semantics of three kinds of basic objects: individuals, sets, and lambda expressions. Informally, the DLOG user writes assertions about these three kinds of objects.

Formally, much of the semantics of the DLOG language can be specified in a straightforward way by using contextual description. However, certain components of the language have an intended semantics whose specification requires a meta language description.

The sentence level syntax of DLOG is nearly identical to the definite clause language of Prolog. This restriction permits the use of Prolog's proof procedure as the foundation of DLOG's proof procedure. DLOG's term syntax, however, is unlike any existing language, and requires special attention from both the semantic and proof-theoretic view. For example, DLOG's proof procedure uses an elaborate matching procedure that extends standard unification with a procedure for directly manipulating the DLOG descriptive terms. Briefly, the DLOG system is realized by compressing the 3-sorted language to a Horn clause syntax. DLOG descriptions are restricted to a term syntax by employing description operators (cf. [Hilbert39, Leisenring69, Robinson79]). The general Prolog proof strategy for Horn clauses can then be used, together with special procedures that extend unification to behave in accordance with an elaborated equality theory.

This chapter specifies the logic of DLOG, and explains how a subset of the language can be mapped to Prolog's definite clause language. Chapter 4 extends the discussion with further details on the motivation for and use of descriptive terms, and chapter 5 discusses an experimental implementation, and chapter 6 speculates on a Prolog-independent foundation for the semantics of DLOG.

3.1. Syntax

The language underlying the DLOG data model is a 3-sorted predicate language, where the sorts are *individuals, sets,* and *lambdas.* The syntax is 3-sorted for user convenience, but one could translate an n-sorted language to a single-sorted language [Wang52] to explain its semantics in terms of a single-sorted theory.

The DLOG alphabet consists of the following symbols.

(1) *strings:* a finite number of characters selected from 0-9, a-z, A-Z, - (hyphen);

(2) *logical constants:* $\forall$ (universal quantification), $\exists$ (existential quantification), ι (Russel's symbol), ϵ (Hilbert's symbol), λ (lambda), $=$ (equality symbol), $\in$ (element-of symbol), $\wedge$ (and symbol), $\vee$ (inclusive or symbol), $\veebar$ (exclusive or symbol), $\supset$ (implies symbol), $\bar{\ }$ (not symbol), { } (braces), ( ) (parentheses), [ ] (square brackets), $< >$ (angle brackets), . (period), : (colon), , (comma),

(3) *individual variables:* $x_1, x_2, x_3, \cdots$

(4) *set variables:* $X_1, X_2, X_3, \cdots$

(5) *lambda variables:* $l_1, l_2, l_3, \cdots$

(6) *individual constants:* any string is an individual constant;

Individual constants are used to name individuals of a domain. Examples of strings are 'CS422', 'Asterix', 'C-3P0', and 'R2D2'.

(7) *predicate constants:* if α is any string and $\tau_1, \tau_2, \cdots, \tau_n$ are sorts, then $<\alpha, \tau_1, \tau_2, \cdots, \tau_n>$ is an n-ary predicate constant.

In a single-sorted language, we can associate a number of arguments with a string (an "arity") to create a predicate constant, and then use that predicate constant to name a relation in the application domain. For example, we might choose the string 'topic-of' to name the binary relation on courses and their topics, e.g., 'topic-of(CS422,AI)'.

In DLOG's 3-sorted language we attribute an arity to a string and a sort to each argument position. For example, if the string 'topic-of' is intended to name a relation on $individuals \times individuals$ then we name that relation with the DLOG predicate constant '<topic-of, individual, individual>'. The tuple defining a predicate constant carries with it information about the sorts on which the predicate is defined. In terms of programming language concepts, a predicate constant is like a procedure declaration whose arguments are all drawn from pre-specified data types.

The DLOG language defined here includes terms that are defined in terms of formulas, so the definition is mutually inductive on terms and formulas. We first define the terms of DLOG, and then the formulas.

3.1.1. Term syntax

The *terms* of DLOG are of three distinct sorts: individual terms, set terms, and lambda terms.

The *individual terms* are

(T1) individual variables,

(T2) individual constants,

(T3) *definite individuals:* if α_1 is an individual variable and $\Phi(\alpha_1)$ is a formula in which α_1 appears free, then $\iota\alpha_1.\Phi(\alpha_1)$ is an individual term called a definite individual;

DLOG's definite individual provides a shorthand syntax for referring to a unique individual whose name is unknown. Intuitively, the variable binding symbol 'ι' can be read as the English definite article "the." For example, we might refer to "the head of Computer Science" as

$$\iota x_1.head-of(x_1,ComputerScience)$$

We normally expect that the variable bound with the symbol 'ι' appears somewhere in the formula that constitutes the body of the description.

(T4) *indefinite individuals:* if α_1 is an individual variable and $\Phi(\alpha_1)$ is a formula in which α_1 appears free, then $\epsilon\alpha_1.\Phi(\alpha_1)$ is an individual term called an indefinite individual.

When we need to refer to "any old α_1" with some property specified by a formula '$\Phi(\alpha_1)$', we can use an indefinite individual. The variable binding symbol 'ϵ' can be read as the English indefinite article "a" or "an." For example, "a course with course number greater than 300" might be referred to by the indefinite individual

$$\epsilon x_1.course(x_1) \wedge [\exists x_2.course-no(x_1,x_2) \wedge x_2 \geq 300]$$

As for definite individuals, we normally expect that the variable bound with the symbol 'ϵ' appears somewhere in the formula that constitutes the body of the description.

The *set terms* are

(T5) set variables,

(T6) *set constants:* if $\alpha_1,\alpha_2, \cdots ,\alpha_n$ are individual constants, then $\{\alpha_1 \wedge \alpha_2 \wedge \alpha_3 \wedge \cdots \wedge \alpha_n\}$ is a set term called a set constant;

A DLOG set constant provides a syntax for naming a finite collection of DLOG individuals.[11] For example, '{CS305 ∧ CS307 ∧ CS309}' names the set consisting of the three domain individuals denoted by the individual constants 'CS305', 'CS307', and 'CS309'. When we want to attribute a property to a set of individuals rather than each of its members, we can use a set constant, e.g., 'cardinality-of({CS305 ∧ CS307 ∧ CS309}, 3)'.

(T7) *definite sets:* if α_1 is an individual variable and $\Phi(\alpha_1)$ is a formula in which α_1 appears free, then $\{\alpha_1{:}\Phi(\alpha_1)\}$ is a set term called a definite set;

A definite set is used to refer to a set consisting of all individuals that satisfy some property. The name "definite" was chosen to mirror its use in "definite individual." As the semantics will later indicate, a definite set is "definite" because it refers to all individuals *in the current database* who satisfy the specified property. For example, "the set of all numerical analysis courses" might be designated as

[11] The ∧ symbol is an abuse of notation, of course, and could be replaced with a more conventional notation. However, the dual use of the the ∧ is retained, to conform with a similar dual use of the $\triangledown$ and ∨ symbols in the definition of indefinite sets.

$\{x_1{:}course(x_1) \wedge topic-of(x_1,NA)\}$

(T8) *indefinite sets:*

(T8.1) if α_1 is an individual variable, α_2 is a set variable, and $\Phi(\alpha_1)$ and $\Psi(\alpha_2)$ are formulas in which α_1 and α_2 occur free, then $\{\alpha_1,\alpha_2{:}\Phi(\alpha_1) \wedge \Psi(\alpha_2)\}$ is a set term called an indefinite set;

(T8.2) if $\alpha_1,\alpha_2, \cdots ,\alpha_n$ are individual constants, and $c_1,c_2, \cdots c_{n-1}$ are symbols such that $c_i \in \{\wedge, \vee, \veebar\}$, for $1 \leq i \leq n-1$ and at least one c_i is $\vee$ or $\veebar$, then $\{\alpha_1 c_1 \alpha_2 c_2 \alpha_3 c_3 \cdots c_{n-1} \alpha_n\}$ is a set term called an indefinite set.

Indefinite sets are "indefinite" in the sense that they refer to one of a set of sets. Like indefinite individuals, they are intended to be used to refer to "any old set" that is an element of *the set of sets* that satisfy the specified properties. For example, the indefinite set

$\{x_1,X_1{:}course(x_1) \wedge cardinality-of(X_1,3)\}$

is the DLOG term that represents an arbitrary set that is a member of the set of "all 3 element sets of courses."

The second form of an indefinite set is for further convenience: in the situation where it is known which individual constants are potential members of the set, the form given by syntax description (T8.2) is appropriate. For example, the indefinite set '{CS115 $\veebar$ CS118}' refers to one of the elements of the set {{CS115}, {CS118}}. When the symbols '$\wedge$', '$\vee$', and '$\veebar$' are used together, parentheses can be used to indicate grouping, e.g., '{(CS115 $\veebar$ CS118) $\wedge$ ENGL100 }' refers to either {CS115, ENGL100} or {CS118, ENGL100}. Without parentheses, '$\wedge$' binds most tightly; '$\vee$' and '$\veebar$' have equal priority, associating left to right; the fully parenthesized form of '{a $\wedge$ b $\veebar$ c $\vee$ d}' is '{((a $\wedge$ b) $\veebar$ c) $\vee$ d}'.

The *lambda terms* are

(T9) lambda variables,

(T10) *lambda constants:* if α is an individual variable and $\Phi(\alpha)$ is a formula in which α is free, then $\lambda\alpha.\Phi(\alpha)$ is a lambda term called a lambda constant.

Lambda constants were introduced to capture a kind of individual occurring naturally in the DDB domain: regulations. For example, to describe a typical degree program we must classify requirements of that program, e.g., "nobody can register if they're under 16 years old" refers to a regulation that uses the lambda constant

$$\lambda x_1.[\exists x_2.age-of(x_1,x_2) \wedge x_2 \geq 16]$$

In this way regulations can be placed in relation to other individuals and sets, e.g.,

$$program-prerequisite(BScMajors, first, \lambda x_1.[\exists x_2.age-of(x_1,x_2) \wedge x_2 \geq 16])$$

says "one of the regulations for the first year of a BScMajors program is that an individual be at least 16 years old."

3.1.2. Formula syntax

The *formulas* of DLOG are defined inductively as follows:

(F1) *atomic formulas:* if $t_1,t_2, \cdots ,t_n$ are terms of sort $\tau_1,\tau_2, \cdots ,\tau_n$ respectively, and $<\alpha,\tau_1,\tau_2, \cdots ,\tau_n>$ is an n-ary predicate constant, then $<\alpha,\tau_1,\tau_2, \cdots ,\tau_n>(t_1,t_2, \cdots ,t_n)$ is an atomic formula;

A DLOG atomic formula is an n-ary predicate constant followed by a parenthesized list of DLOG terms of the appropriate sort. As explained above, the predicate constants in a multi-sorted language are slightly more complex, as they must encode their arity and the sort of each argument. In most of our examples, we write only the string of the predicate constant instead of the complete tuple; the sort of each argument should be obvious from the context. For example, the atomic formula

$$\begin{aligned} program-prerequisite(&BScMajors, \\ &\{third \wedge fourth\}, \\ &\lambda x_1.completed(x_1,CS315)) \end{aligned}$$

would be more formally written as

$$\begin{aligned} <program-prerequisite,individual,set,lambda>(&BScMajors, \\ &\{third \wedge fourth\}, \\ &\lambda x_1.completed(x_1,CS315)) \end{aligned}$$

Intuitively, this atomic formula names an instance of a relation that says "it is a program requirement for a BScMajors that a student have completed CS315 sometime during years three and four."

(F2) *formulas:*

(F2.1) if α is an atomic formula, then both α and $(\neg\alpha)$ are formulas;

(F2.2) if α_1 and α_2 are formulas, then so are $(\alpha_1 \wedge \alpha_2)$, $(\alpha_1 \vee \alpha_2)$, $(\alpha_1 \triangledown \alpha_2)$, and $(\alpha_1 \supset \alpha_2)$,

(F2.3) if α_1 is a formula, then $(\forall x_1.\alpha_1)$, $(\forall X_1.\alpha_1)$, $(\forall l_1.\alpha_1)$, $(\exists x_1.\alpha_1)$, $(\exists X_1.\alpha_1)$, and $(\exists l_1.\alpha_1)$ are all formulas.

For formulas constructed with the operator symbols '$\neg$', '$\wedge$', '$\vee$', and '$\veebar$', the parentheses may be omitted, in which case operator precedence is as for their use in the indefinite sets of syntax description (T8.2); the only difference is the addition of '$\neg$', which is unary, and will bind more tightly than all other operators.

3.2. Semantics

The method of contextual definition will provide a specification of the intended meaning of most of the complex terms, however, lambda terms require special treatment. We first provide the contextual definition for individual and set descriptions. Then follows a brief description of an interpretation for the DLOG language. A later section (§3.5) describes the manipulation of λ constants at the meta level.

3.2.1. Contextual definitions of DLOG descriptions

The semantics of DLOG descriptions can be defined by the method of contextual definition. This style of defining the meaning of descriptive terms was employed by Russell (e.g., see [Kaplan75]), and is essentially what one might explain as a "macro definition." The most familiar contextual definition is that, proposed by Russell, for definite individuals. He proposed that sentences of the form

$$\Phi(\iota x_1.\Psi(x_1))$$

be viewed as an abbreviation for

$$\exists x_1.[\Phi(x_1) \wedge \forall x_2.[\Psi(x_2) \equiv x_2 = x_1]]$$

and argued that if the latter sentence was not true in the situation being considered, then the description '$\iota x_1.\Psi(x_1)$' used in the former sentence had no denotation. Philosophical debate on the meaning of descriptive terms continues to this day; further relevant discussion is postponed to chapter 4.

The contextual definition of DLOG's individual and set descriptions are as follows:

$$\Phi(\iota x_1.\Psi(x_1)) \equiv \exists x_1.[\Phi(x_1) \wedge \forall x_2.[\Psi(x_2) \equiv x_2 = x_1]] \tag{3.1}$$

$$\Phi(\epsilon x_1.\Psi(x_1)) \equiv \exists x_1.\Phi(x_1) \wedge \Psi(x_1) \tag{3.2}$$

$$\Phi(\{x_1 : \Psi(x_1)\}) \equiv \exists X_1.[\Phi(X_1) \wedge \forall x_1.[\Psi(x_1) \equiv x_1 \in X_1]] \tag{3.3}$$

$$\Phi(\{x_1, X_1 : \Psi(x_1) \wedge \Omega(X_1)\}) \equiv \exists X_1.[[\Phi(X_1) \wedge \forall x_1.[x_1 \in X_1 \supset \Psi(x_1)]] \wedge \Omega(X_1)] \tag{3.4}$$

$$\Phi(\{\alpha_1 \wedge \alpha_2\}) \equiv \exists X_1.[\Phi(X_1) \wedge \alpha_1 \in X_1 \wedge \alpha_2 \in X_1 \wedge \forall x_1.[x_1 \in X_1 \supset x_1 = \alpha_1 \vee x_1 = \alpha_2]] \tag{3.5}$$

$$\Phi(\{\alpha_1 \vee \alpha_2\}) \equiv \exists X_1.[\Phi(X_1) \wedge [X_1 = \{\alpha_1 \wedge \alpha_2\} \vee X_1 = \{\alpha_1\} \vee X_1 = \{\alpha_2\}]] \tag{3.6}$$

$$\Phi(\{\alpha_1 \veebar \alpha_2\}) \equiv \exists X_1.[\Phi(X_1) \wedge [X_1 = \{\alpha_1\} \veebar X_1 = \{\alpha_2\}]] \tag{3.7}$$

These contextual definitions provide a way of interpreting the meaning of descriptive terms by interpreting their definitions in a standard language. Intuitively, this style of definition reveals the descriptive objects to be artifacts of the language syntax — there meaning is determined solely in terms of their definition.

3.2.2. Interpretation of formulas

An interpretation **I** of a DLOG theory **DB** consists of

(I1) a non-empty domain $D = \{D_I \cup D_S \cup D_R\}$, where D_I is a non-empty domain of individuals over which the individual variables range, $D_S = \{S : S \subseteq D_I\}$ over which the set variables range, and D_R is a set of *regulations* over which the lambda variables range;

(I2) for each individual constant i of **DB**, an assignment $\mathbf{V}(i) \in D_I$; for each set constant s of **DB**, an assignment $\mathbf{V}(s) \in D_S$; and for each lambda constant l of **DB**, an assignment $\mathbf{V}(l) \in D_R$;

(I3) for each n-ary predicate constant, except '=', $<\alpha, \tau_1, \tau_2, \cdots, \tau_n>$ an assignment $\mathbf{V}(<\alpha, \tau_1, \tau_2, \cdots, \tau_n>) = \mathbf{R}$, where **R** is a relation on $\mathrm{T}(\tau_1) \times \mathrm{T}(\tau_2) \times \cdots \times \mathrm{T}(\tau_n)$, such that $\mathrm{T}(\mathit{individual})$ is D_I, $\mathrm{T}(\mathit{set})$ is D_S, and $\mathrm{T}(\lambda)$ is D_R;

(I4) an assignment to the predicate constant '=' of the identity relation in D; in particular, of $<=, \mathit{individual}, \mathit{individual}>$ to identity in D_I, mutis mutandis for D_S and D_R; and

(I5) an assignment to the predicate constant '$\in$' of the set membership relation in D; in particular, of $<\in, \mathit{individual}, \mathit{set}>$ to set membership in D.

We now define satisfiability for DLOG formulas.

Let F be any atom of the form $<\alpha, \tau_1, \tau_2, \cdots, \tau_n>(t_1, t_2, \cdots, t_n)$ where each t_i is of sort τ_i for $1 \leq i \leq n$; then F is satisfied by an interpretation **I** if and only if the tuple $<\mathbf{V}(t_1), \mathbf{V}(t_2), \cdots, \mathbf{V}(t_n)>$ is in the relation $\mathbf{V}(<\alpha, \tau_1, \tau_2, \cdots, \tau_n>)$ in **I**.

The logical constants '$\forall$', '$\exists$', '$\neg$', '$\wedge$', '$\vee$', and '$\veebar$' have the usual interpretation:

($\forall$) if F is any formula and x is any variable, then $(\forall x.F)$ is true in I if and only if F is true in I for all assignments $V(x) \in D$;

($\exists$) if F is any formula and x is any variable, then $(\exists x.F)$ is true in I if and only if F is true in I for at least one assignment $V(x) \in D$;

($\neg$) if F is any atom, then $(\neg F)$ is true n I if and only if F is not true in I;

($\wedge$) if F_1 and F_2 are any two formulas, then $(F_1 \wedge F_2)$ is true in I if and only if F_1 and F_2 are true in I;

($\vee$) if F_1 and F_2 are any two formulas, then $(F_1 \vee F_2)$ is true in I if and only if F_1 or F_2 are true in I;

($\veebar$) if F_1 and F_2 are any two formulas, then $(F_1 \veebar F_2)$ is true in I if and only if either i) F_1 and $(\neg F_2)$ are true in I, or ii) $(\neg F_1)$ and (F_2) are true in I.

3.3. An implementable subset of DLOG

To demonstrate that the expressive power of DLOG is actually useful requires an implementation that supports the development of a real DLOG database. This requires a proof procedure that will manipulate the language defined in §3.1 in a way that is consistent with the semantics described in §3.2.

The strategy will be to restrict the language defined in §3.1 so that a proof procedure based on Prolog's SLD resolution can be used as the foundation of an implementation. This foundation will be embellished with special procedures to manipulate DLOG's complex terms. The objective is an implementation that combines the expressiveness of DLOG with the relative efficiency of Prolog. This can be done by placing syntactic restrictions on the DLOG syntax of §3.1, thus compressing it to a Horn clause syntax augmented with DLOG's complex terms.

The following section describes the restriction to a Horn clause syntax, and the subsequent section describes the extension of Prolog's SLD resolution to manipulate DLOG's complex terms. The extension uses Prolog's negation-as-failure [Clark78] to manipulate negative information, together with an extended definition of equality that is used to define DLOG unification. The DLOG unification procedure can recursively invoke the proof mechanism to determine the equality of complex terms.

Subsequent sections will include further modifications of the Prolog proof procedure to manipulate lambda terms, and apply constraints as assertability rules.

3.3.1. Syntax restrictions

The major syntax restrictions include the standard Prolog restriction to Horn clauses (e.g., see [Kowalski79]), and a further restriction that will allow description bodies to be interpreted as DLOG queries.

The Horn clause restriction requires that every formula, when in clausal form, have *at most* one positive literal. This gives us three general syntactic forms for formulas. Let α_i, $1 \leq i \leq n$, be atomic formulas, as defined in §3.1, syntax description F1. Let x_1, x_2, ..., x_m be the free variables in the α_i. Then a formula of the general form

$$\forall x_1 x_2 \cdots x_m \alpha_1 \vee {}^{\sim}\alpha_2 \vee \cdots \vee {}^{\sim}\alpha_n$$

can be written as

$$\forall x_1 x_2 \cdots x_m \alpha_1 \subset \alpha_2 \wedge \cdots \wedge \alpha_n \tag{3.8}$$

and is called a *rule*. A formula consisting of only one positive literal, viz.

$$\forall x_1 x_2 \cdots x_m \alpha_i \tag{3.9}$$

is called a *fact*. Finally, a sentence of the form

$$\forall x_1 x_2 \cdots x_m {}^{\sim}\alpha_1 \vee {}^{\sim}\alpha_2 \vee \cdots \vee {}^{\sim}\alpha_n$$

can be rewritten as

$$^{\sim}\exists x_1 x_2 \cdots x_m \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n \tag{3.10}$$

and is called a *query*. Notice that the symbol '${}^{\sim}$' does not appear in the formulas (3.8) and (3.9). Furthermore, since all free variables are treated uniformly, the quantifier prefix can be dropped from all forms; free variables in facts and rules are then assumed to be universally quantified, and those in queries are assumed to be existentially quantified. In fact, in Prolog query syntax, it is customary to drop both the negation sign '${}^{\sim}$' and the existential quantifier prefix; all free variables in queries are assumed to be existentially quantified.

To these formula schemas we add the logical constants '$\vee$', '$\forall$' and '$\neg$' to complete our restricted language. Notice that the newly added symbol '$\neg$' is different from the symbol '${}^{\sim}$'; the former will be used below to denote "negation-as-failure," while the latter is the standard logical constant, as used in §3.1, and above to define the Horn clause restriction. Notice also that the symbols '$\vee$' and '$\veebar$' are identical to those appearing in the more general syntax (§3.1), however they have restricted interpretations in the subset being defined here. In particular, the exclusive or symbol '$\veebar$' will be defined in terms of negation-as-failure. Similarly, the predicate constants '$\in$' and '$=$' will be given restricted interpretations.

Before redefining the syntax description F2 (§3.1) to conform to the structure of the Horn clause schemas, one intermediate syntactic item is necessary. A *body* is defined as follows:

(B1) bodies:

(B1.1) if α is an atomic formula (cf. syntax description F1, §3.1), then α is a body;

(B1.2) if α_1 and α_2 are bodies, then so are $(\neg\alpha_1)$ $(\alpha_1 \wedge \alpha_2)$, $(\alpha_1 \vee \alpha_2)$, $(\alpha_1 \veebar \alpha_2)$.

The definition of the syntactic component "body" allows the completion of the restricted DLOG language definition. First, the syntax descriptions of complex terms (T3, T4, T7, T8.1, and T10 of §3.1) are replaced with the following:

(T3) *definite individuals:* if α_1 is an individual variable and $\Phi(\alpha_1)$ is a **body** in which α_1 appears free, then $\iota\alpha_1.\Phi(\alpha_1)$ is an individual term called a definite individual;

(T4) *indefinite individuals:* if α_1 is an individual variable and $\Phi(\alpha_1)$ is a **body** in which α_1 appears free, then $\epsilon\alpha_1.\Phi(\alpha_1)$ is an individual term called an indefinite individual.

(T7) if α_1 is an individual variable and $\Phi(\alpha_1)$ is a **body** in which α_1 appears free, then $\{\alpha_1:\Phi(\alpha_1)\}$ is a set term called a definite set;

(T8.1) if α_1 is an individual variable, α_2 is a set variable, and $\Phi(\alpha_1)$ and $\Psi(\alpha_2)$ are bodys in which α_1 and α_2 occur free, then $\{\alpha_1,\alpha_2:\Phi(\alpha_1)\wedge\Psi(\alpha_2)\}$ is a set term called an indefinite set;

Consistent with the Prolog convention explained above, all free variables within descriptions will be assumed to be existentially quantified.

Finally, we redefine the syntax description F2 for formulas.

(F2) *formulas:*

(F2.1) if α is an atomic formula, then α is a formula called a fact;

(F2.2) if α_1 is an atomic formula and α_2 is a body, then $(\alpha_1 \subset \alpha_2)$ is a formula called a rule;

(F2.3) if α_1 is an atomic formula and α_2 is a body, then $(\alpha_1 \supset \alpha_2)$ is a formula called a constraint;

(F2.4) if α is a body, then $?\alpha$ is a formula call a query.

The meaning of constraints, as sentences of a DLOG database, is given by the standard interpretation of DLOG (§3.2.2). Their meaning as integrity constraints is described in §3.6.

3.4. DLOG derivability

To complete the specification of the logic component of the DLOG data model we need to specify what it means for a DLOG query Q to be derivable from a set of DLOG assertions DB (i.e., a collection of assertions). We need a specification for the relation usually called '$\vdash$', so that the expression $DB \vdash Q$ is meaningful. Henceforth we will rename the symbol '$\vdash$' as 'derivable', and write

$$derivable(DB,Q)$$

to mean that query Q follows, by some number of inferences, from database DB. Ultimately we desire an implementation for the derivable relation that will allow us to determine, given DB and Q, if $derivable(DB,Q)$.

In describing the relation *derivable*, we are anticipating a particular style of implementation based on Prolog's SLD resolution [Lloyd82]. Our strategy for interpreting DLOG's complex terms is similar to others (e.g., [Pietrzykowski73, Huet73]) in that the focus is on providing the necessary extensions to unification. However, in contrast to Pietrzykowski and Huet, we take a more ad hoc approach. Because the intended interpretation of DLOG theories is restricted to finite domains, the necessary unification extensions can be made by specifying equality axioms for the DLOG complex terms, and then interpreting these axioms as a Prolog program that implements DLOG unification.

We might have attempted a more traditional specification of derivability in terms of axioms and rules of inference (e.g., [Rogers71, Mendelson64]), or simply adopted a general resolution proof procedure (e.g., [Loveland78, Robinson79]). but either approach would still require further attention to the manipulation of DLOG's complex terms. The Prolog foundation was selected because it is a practical implementation of a proof procedure. The extended behaviour of the Prolog proof procedure is achieved by using Prolog to execute an appropriate meta level description of how to manipulate DLOG terms.

3.4.1. Equality theories for Prolog

The SLD proof procedure implemented in Prolog uses a depth-first left-to-right selection strategy that attempts to derive queries by matching them against facts and rules in the database. For example, an atomic query

$$\Phi(t_1,t_2, \cdots t_n)$$

can be reduced to another query

$$\Psi(s_1,s_2, \cdots s_m)\theta$$

by backchaining through the rule

$$\Phi(u_1,u_2, \cdots u_n) \subset \Psi(s_1,s_2, \cdots s_n)$$

This reduction is legal only when '$\Phi(t_1,t_2, \cdots t_n)$' matches '$\Phi(u_1,u_2, \cdots u_n)$' with most general unifier θ, as determined by the unification algorithm [Robinson65]. When a query of the form $\Phi(t_1,t_2, \cdots t_n)$ *unifies* with a fact of the form $\Phi(s_1,s_2, \cdots s_n)$, a proof is complete (i.e., the query is successful).

The unification operation shows how two terms can be made equal. To understand how Prolog implements equality, we can use Clark's [Clark78] "general form": assertions of the form $\Phi(t_1,t_2, \cdots t_n)$ where t_i, $1 \leq i \leq n$ are terms, can be rewritten as

$$\Phi(x_1,x_2, \cdots x_n) \subset x_1=t_1 \wedge x_2=t_2 \wedge \cdots \wedge x_n=t_n$$

and implications $\Phi(t_1,t_2, \cdots ,t_n) \subset \Psi_1 \wedge \Psi_2 \wedge \cdots \wedge \Psi_m$ can be rewritten as

$$\Phi(x_1,x_2, \cdots ,x_n) \subset x_1=t_1 \wedge x_2=t_2 \wedge \cdots \wedge x_n=t_n \wedge \Psi_1 \wedge \Psi_2 \wedge \cdots \wedge \Psi_m$$

where the x_i, $1 \leq i \leq n$ are new variables not occuring in the original formulas.

Clark's general form shows how the unification algorithm is simply an efficient way to derive the necessary equalities $x_i=t_i$, $1 \leq i \leq n$. He combines this general form with equality axioms to specify how a definite clause database can be "completed" to explain the logical basis for Prolog's negation-as-failure: Prolog will answer "yes" to a query '$\neg\Phi$', just in case it finds that all derivation search paths for a proof of 'Φ' end with failure. Clark's results show how negation-as-failure in the original definite clause database corresponds to deduction in the completed database.

Clark's results also contirubte to the specification of DLOG. First, the DLOG negation symbol '$\neg$' is implemented as Prolog's negation-as-failure. Second, Clark's general form provides a way to explain how the DLOG derivable relation manipulates descriptive terms. Normally, the equalities that sanction query reduction involve only simple individual terms. However, the equalities arising in the general form of DLOG assertions are drawn from a much richer collection of objects. One can therefore view DLOG's derivable relation as an extension of Prolog, but with a different theory of equality. This is what van Emden and Lloyd claim about Colmeraurer's Prolog II [Emden84], which employs a "homogeneous form" to isolate equations, just as Clark does with his general form. In fact, van Emden and Lloyd show that Prolog without negation-as-failure uses only the equality axiom '$\forall x.x=x$'. This isolation of the equality theory from the general proof procedure is also used by Kornfeld [Kornfeld83], who extends a LISP-based Prolog system with equality axioms.

It should be noted here that it is appealing to extend unification by enriching Prolog's theory of equality, but that it is easy to destroy the unification algorithm's termination property. In general, this is because one can extend the definition of the predicate '=' until it has the power of a general functional model of computation. For example, Goguen and Meseguer [Goguen84] provide such a definition of Horn clause logic with equality and explain many of the problems of computing within such a theory. Note also that the equality theory of Colmeraurer's Prolog II is not relevant to DLOG, as DLOG's language does not admit infinite terms.

The DLOG extensions to unification described below are more conservative than those of Goguen and Meseguer in that only those equality axioms for DLOG's complex terms are added. These axioms are not used as general rewrite rules, but as Prolog axioms that define a restricted version of the '=' relation.

3.4.2. DLOG's equality theory

The new equalities to be handled by DLOG unification are those that could be possibly introduced by rewriting DLOG assertions and implications in Clark's general form. We first require the equality axioms implicit in Prolog (with negation-as-failure): if $\{\alpha_1,\alpha_2, \cdots ,\alpha_n\}$ are the individual constants in a DLOG database, then we have

$$\alpha_i \neq \alpha_j \text{ for } i \neq j \quad 1 \leq i,j \leq n \qquad \text{(name uniqueness)}$$

This is the same set of name uniqueness axioms used by Reiter [Reiter83] in his reconstruction of relational databases (see §2.2.3). In DLOG unification, these axioms are manifest in the way an implementation determines how two individual constants are different: by comparing machine representations of strings.

The remaining axiom schemas required by Clark are substitution, and those that define '=' as an equivalence relation:

$$\forall x.[x=x] \qquad \text{(reflexive)}$$
$$\forall xy.[x=y \supset y=x] \qquad \text{(symmetric)}$$
$$\forall xyz.[x=y \wedge y=z \supset x=z] \qquad \text{(transitive)}$$
$$\forall v_1,v_2, \cdots ,v_n,w_1,w_2, \cdots ,w_n.\Phi(v_1,v_2, \cdots ,v_n) \qquad \text{(substitution)}$$
$$\wedge v_1=w_1 \wedge v_2=w_2 \wedge \cdots \wedge v_n=w_n \supset \Phi(w_1,w_2, \cdots ,w_n)$$

where v_i,w_i are individual variables, as appropriate for any n-ary predicate constant Φ.

We now require axiom schema that specify the equality of DLOG's descriptive terms. Part of the task is simplified because the sorts of DLOG are distinct.[12] Therefore we need only consider the possible combinations of equality expressions within each sort.

In addition to the equality axioms given above, we need the following:

$$x=\epsilon y.\Phi(y) \equiv [\Phi(x)] \qquad (3.15)$$

This axiom schema says that to demonstrate the equality $x=\epsilon y.\Phi(y)$, prove that x satisfies $\Phi(x)$. As an instruction to DLOG unification, it says that when matching atoms of the form

[12] The implementation of chapter 5 actually relaxes this distinction to streamline the user interface, and to increase the implementation's efficiency. See chapter 5, §5.2.3.

$$\Phi(t_1,t_2,\cdots,t_i,\cdots,t_n)$$

and

$$\Phi(t_1,t_2,\cdots,t_{i-1},\epsilon x.\Psi(x),t_{i+1},\cdots,t_n)$$

check that $\Psi(t_i)$ is derivable.

The remaining axiom schemas for individual descriptions provide similar instructions for DLOG unification:

$$x=\iota y.\Phi(y)\equiv[\forall z.[\Phi(z)\equiv z=x]] \tag{3.16}$$

$$\epsilon x.\Phi(x)=\iota y.\Psi(y)\equiv[\exists u.[\forall z.\Psi(z)\equiv u=z]\wedge\Phi(u)] \tag{3.17}$$

$$\epsilon x.\Phi(x)=\epsilon y.\Psi(y)\equiv[\exists z.[\Phi(z)\equiv\Psi(z)]] \tag{3.18}$$

$$\iota x.\Phi(x)=\iota y.\Psi(y)\equiv[\exists u\,\forall z.[\Phi(z)\equiv\Psi(z)]\equiv z=u] \tag{3.19}$$

The next sort to consider is sets. DLOG set theory is simple, since it is constructed from the finite set of DLOG individuals. The equality axioms for individuals together with the 'ϵ' (element of) relation allow us to specify the following axiom for set equality:

$$\forall X_1X_2.X_1=X_2\equiv[\forall x.x\,\epsilon X_1\equiv x\,\epsilon X_2] \tag{3.20}$$

This specifies the conditions under which sets are equal. Since the closed world assumption is in force, the universal quantifier '$\forall x$' is interpreted over a finite domain of individuals, and set equivalence can be determined extensionally. In fact the equality of all sets, including set descriptions, could be determined by using this axiom, as long as the 'ϵ' relation was defined on set descriptions. But determining the equality of set descriptions by comparing their extensions, though possible in this finite case, would be very inefficient. However, the existence of set descriptions provides a more computationally efficient way of determining set equality. For DLOG, the closed world assumption makes intensional and extensional equivalence the same; so alternative computational strategies for determining set equivalence are available for set descriptions:

$$\{x:\Phi(x)\}=\{y:\Psi(y)\}\equiv[\forall z.\Phi(z)\equiv\Psi(z)] \tag{3.21}$$

$$\{x,X:\Phi(x)\wedge\Psi(X)\}=\{y,Y:\Omega(y)\wedge\Pi(Y)\}\equiv\exists Z.[\forall z.z\,\epsilon Z\supset\Phi(z)\wedge\Psi(z)]\wedge\Omega(Z)\wedge\Pi(Z) \tag{3.22}$$

$$\{x:\Phi(x)\}=\{y,Y:\Psi(y)\wedge\Pi(Y)\}\equiv[\exists Z.[\forall z.z\,\epsilon Z\equiv\Phi(z)]\wedge\Pi(Z)\wedge\forall z.\Phi(z)\supset\Psi(z)] \tag{3.23}$$

Axiom schema (3.21), (3.22), and (3.23) explain how a theorem prover would attempt to demonstrate the equality of set descriptions "intensionally," by proving that their definitions are logically equivalent. For DLOG, the difference will be computational efficiency; when the closed world assumption is not in force, there are many well known situations where the methods are not equivalent.[13]

Finally, the equality of lambda constants is specified by

$$\lambda x_1.\Phi(x_1)=\lambda x_2.\Psi(x_2)\equiv[\forall x.\Phi(x)\equiv\Psi(x)] \tag{3.24}$$

Notice that new logical constants '$\in$' (element-of) and '$\equiv$' (equivalence) have been introduced to describe how the equality of various complex terms can be established. Definitions of these constants suitable for the restricted DLOG language (§3.2.1) are defined as Prolog programs, and are explained in more detail in chapter 5 (§5.2.3.1).

3.4.3. Extending Prolog derivability for DLOG

The equality terms in Clark's general form never actually appear in any derivation of a query, in fact the equality symbol '=' isn't normally included in the Prolog language (cf. [Emden84]). Instead, the required derivations of the equalities $x_i=t_i$, $1 \leq i \leq n$ are performed implicitly by the unification algorithm. Because unification will establish the necessary equalities, and bind constants to variables where appropriate, the algorithm for Prolog derivations can be specified very succinctly (e.g., see [Bowen82]). For example, the following is a brief description of Prolog derivability in Prolog:

P1) *derivable(DB, Q)* $\subset$
    axiom(DB, S) $\wedge$ *unify(Q, S).*

P2) *derivable(DB, Q)* $\subset$
    axiom(DB, S $\subset$ *R)* $\wedge$ *unify(Q, S)* $\wedge$ *derivable(DB, R).*

P3) *derivable(DB, Q* $\wedge$ *R)* $\subset$
    derivable(DB, Q) $\wedge$ *derivable(DB, R).*

The variable *DB* is a Prolog "program," or set of assertions. Briefly, the three sentences can be read as follows:

p1) an atomic query *Q* is derivable from a database *DB* if *S* is an assertion in database *DB* (as determined by *axiom*), and *Q* and *S* unify;

[13] For the logical background see [Quine80, chapter VIII, *Reference and Modality*]; for examples of practical considerations see [Schubert76, §8] or [Moore76, §§1, 3].

p2) an atomic query Q is derivable from a database DB if S is a rule in database DB (as determined by *axiom*), Q and S unify, and the new query R is derivable in DB; and

p3) a conjunction of atomic queries is derivable from DB, if both the first conjunct and the remainder of the query is derivable.

Here the unify operation does most of the work for Prolog, but this is not the case for DLOG. Unification will not establish the equality of DLOG's complex terms. Rather than augment the restricted DLOG language with '=' and other symbols introduced in the definition of complex terms (e.g., 'ϵ', '$\blacksquare$'), we can retain the simplicity of the Prolog proof strategy by providing an augmented "unification"[14] algorithm that will establish the necessary equalities.

The behaviour of DLOG unification can be described in a way similar to the way descriptive terms were explained by contextual definition: the above equality axiom schema that express the conditions under which two DLOG terms are equal. These schema introduce rather elaborate specifications of how the equality of descriptive terms can be proved, and their manipulation would, in general, require a language in which symbols like 'ϵ' and '$\blacksquare$' were defined. However, instead of explicitly providing DLOG with a more powerful theorem-prover, the equality specifications are implemented as Prolog programs and embedded in DLOG unification. By substituting DLOG unification for the Prolog version, the above description of Prolog derivability provides the basis for DLOG derivability. In a similar way, DLOG derivability can be specified as follows. As above, we will write

derivable(DB,Q)

to mean that DLOG query Q follows, by some number of inferences, from DLOG database DB. The simple specification above then becomes:

D1) $derivable(DB, Q) \subset$
$\quad atom(Q) \wedge$
$\quad axiom(DB, F) \wedge$
$\quad fact(F) \wedge$
$\quad unify(Q, F).$

[14] Henceforth the phrase "DLOG unification" will be used to refer to DLOG's procedure for matching the atoms of queries with assertions or the consequents of implications. In introducing the idea here, the quotes acknowledge that DLOG unification is not the same as the well-known unification algorithm, but is nonetheless very closely related in it's intended use and operation.

D2) *derivable(DB, Q)* ⊂
atom(Q) ∧
axiom(DB, R) ∧
rule(R, H, T) ∧
unify(Q, H) ∧
derivable(DB, T).

D3) *derivable(DB, (Q1* ∨ *Q2))* ⊂
derivable(DB, Q1).

D4) *derivable(DB, (Q1* ∨ *Q2))* ⊂
derivable(DB, Q2).

D5) *derivable(DB, (Q1* ⊻ *Q2))* ⊂
derivable(DB, Q1) ∧
derivable(DB, (¬Q2)).

D6) *derivable(DB, (Q1* ⊻ *Q2))* ⊂
derivable(DB, (¬Q1)) ∧
derivable(DB, Q2).

D7) *derivable(DB, (¬Q))* ⊂
¬ *derivable(DB, Q).*

D8) *derivable(DB, (Q1* ∧ *Q2))* ⊂
derivable(DB, Q1) ∧
derivable(DB, Q2).

The first two axioms (D1, D2) define the top level of DLOG derivability for an atomic query Q (see §3.2.1, syntax definition F2.3). The relation '*atom*(Q)' holds whenever its argument Q is a DLOG atomic query. The relation '*axiom*(DB,F)' holds whenever F is an axiom of the DLOG database DB. In both D1 and D2, the axiom relation serves as a generator of DB axioms potentially relevant to the query Q. The relations '*fact*(F)' and '*rule*(R,H,T)' are used to distinguish DLOG facts and rules. The latter not only verifies that R is a rule, but that H and T are the rule's head (consequent) and tail (antecedent), respectively. For example, the following instance of the *rule* relation holds:

rule(p(x) ⊂ *q(x), p(x), q(x))*

Both D1 and D2 rely on the DLOG unify relation to sanction the continuation of the derivation. The relation '*unify*(Q,F)' holds whenever the atom Q unifies with the atom F. Of course, this DLOG unify relation is not nearly so straightforward as standard unification — it is the key to DLOG derivability.

The remaining axioms (D3-D8) specify the treatment of the DLOG logical constants '$\wedge$', '$\vee$', '$\forall$', and '$\neg$'. Although the derivation of '$\neg$*derivable*(DB,Q)' is generally undecidable, axiom D7 is necessary in that it defines negation-as-failure, as previously mentioned (§3.2.1); the implementation will rely on the standard implementation of negation-as-failure (chapter 5, §5.1.1). The implementation of the DLOG derivable relation in chapter 5 (§5.2.3.1) is a Prolog program that conforms to these specifications.

3.5. Interpretation of lambda constants as regulations

The interpretation of DLOG formulas specified in §3.2.2 interpreted lambda constants as regulations. That is, even though lambda constants have the appearance of predicate abstractions, their first order interpretation is that they are names for regulations.

Used in this way, DLOG lambda constants provide the user with a method of asserting axioms about unary predicate abstractions, intuitively interpreted as regulations. These terms are useful in a variety of ways: viewed as a constant, the assertion

$$\Phi(\lambda x_1.\Psi(x_1))$$

can be interpreted simply as asserting that the property Φ is true of the regulation named by $\lambda x_1.\Psi(x_1)$. These terms are useful because they allow a user to assert relations about properties.

As a unary predicate abstraction, another of their uses can be explained with the aid of a meta relation called *satisfies*. Syntactically, *satisfies* is a two place DLOG predicate constant whose first argument is of the sort *individual* and whose second argument is of the sort λ (as in §3.1.1).

In DLOG, we use an axiom that relates *satisfies* to the DLOG meta predicate *derivable* (§3.4) to provide a method for testing whether an individual satisfies the regulation denoted by a DLOG lambda constant. The definition of *satisfies* is

$$\forall x_1.satisfies(x_1,\lambda x_2.\Psi) \equiv derivable(DB,[x_1/x_2]\Psi)$$

where the notation $[c_1/x_2]\Psi$ denotes the result of substituting term x_1 for each occurrence of the variable x_2 in the body of the lambda term Ψ (e.g., see [Stoy77, ch. 5]). Then, for any individual constant α of a DLOG database DB, $satisfies(\alpha,\lambda x_1.\Psi)$ is true under a DLOG interpretation **I** if and only if

$[\alpha/x_1\Psi]$ is true under **I**. An assertion of the form

$$satisfies(\alpha,\lambda x_1.\Phi) \tag{4.1}$$

is interpreted to mean that, in the current database,

$$\Phi(\alpha) \tag{4.2}$$

is derivable. Indeed (4.1) is a clumsy alternative to (4.2), but by providing lambda constants as terms, we not only provide a way of asserting axioms about regulations, but also a way of using those regulations in question answering. The *satisfies* predicate provides the mechanism for applying lambda constants as unary predicates of the current database.

The intended use of lambda constants is further discussed in §4.1.3.

3.6. DLOG database maintenance

The DLOG language is designed for describing domains that are perceived as individuals, sets of individuals, regulations, and relationships among objects of those sorts. Intuitively, DLOG axioms, queries and assertions are objects too, so we should be able to specify some theory that captures the relationships among those objects. This requires a language for writing axioms — a logic — to precisely define DLOG database maintenance.[15]

The major role of this section is to complete the specification of the DLOG data model by specifying the meaning of assertability in a DLOG database. As a final elaboration of the DLOG data model, we define assertability and specify a syntax by which users can assert integrity constraints for DLOG databases. DLOG constraints provide a way of making invariant assertions about DLOG database relations, and the definition of DLOG assertability provides a method for automatically enforcing constraints to determine the assertability of a database update.

Again, as discussed in §2.3.1, the specification of DLOG database maintenance here does not treat updates that change the data, but rather only those that consistently augment it. As desirable as such a specification is, its development here would be a digression. A major part of such a

[15] See [Borgida81, Mylopoulos80] for an approach based on a procedural representation.

specification can be viewed as a classical database design problem in the sense that one could promote moments of time to the status of individuals, and use a state representation (e.g., see [Kowalski79]) to form the logical foundation of DLOG database maintenance. However, the computational problems associated with such a foundation are significant.

The syntax of DLOG constraints was given in §3.3.1, syntax definition F2.3:

if α_1 is an atomic formula and α_2 is a body, then $(\alpha_1 \supset \alpha_2)$ is a formula called a constraint;

All free variables are assumed to be universally quantified. The following example illustrates a typical constraint of the DDB domain (see §2.3.2).

$$\begin{aligned} &course-enrolled(x_1,x_2) \supset \\ &\qquad student(x_1) \\ &\qquad \wedge course(x_2) \\ &\qquad \wedge registered(x_1,x_3) \\ &\qquad \wedge program-contribution(x_3,x_2) \\ &\qquad \wedge prerequisites(x_2,X_1) \\ &\qquad \wedge satisfied(x_1,X_1) \end{aligned}$$

This constraint says "if x_1 is enrolled in x_2, then x_1 must be a student, x_2 a course, and furthermore, there must be a program x_3 in which x_1 is registered, the course x_2 must contribute to that program x_3, and the student x_1 must have satisfied the set of appropriate prerequisites." Informally, we say that the constraint would be applied whenever an update of the database was attempted. For example, a new assertion '$course-enrolled(Kari,CS442)$' would be rejected if any of the conditions of the above constraint were violated.

To be more formal, we could use the idea of Bowen and Kowalski [Bowen82], and endeavour to axiomatize the relationship between DLOG application databases, assertions, queries, and constraints *within* DLOG. By carefully naming DLOG sentences, we can specify a portion of the DLOG data model within the DLOG language. This means that each DLOG specification would include a DLOG theory that axiomatizes DLOG database maintenance. In the following informal description we assume at least a partial amalgamation of the kind prescribed by Bowen and Kowalski. The specification here is written in terms of a sorted first order language. Chapter five describes how these specifications are used as the basis of a implementation.

The specification of DLOG database maintenance assumes *representability* [Bowen82] of the following predicates:

derivable(X,x)
DLOG formula x is derivable from DLOG database X;

assertable(x,X,Y)
DLOG formula x is asserted as an axiom in the current database X, giving the new database Y;

constraint(x)
DLOG formula x is a DLOG constraint, as specified above;

assertion(x)
DLOG formula x is a DLOG fact or rule, as specified in §3.3.1;

query(x)
DLOG formula x is a DLOG query, as specified in §3.3.1, and

relevant(x,y)
DLOG constraint x is relevant to DLOG assertion y.

Representability means that we expect every true positive instance of a relation to be derivable, but not each negative instance. For example, when Q is a DLOG query, we expect that

$$derivable(DB,Q)$$

is derivable whenever Q is derivable from DB, but we do not require that

$$\neg derivable(DB,Q)$$

be derivable whenever Q is not derivable from DB. Bowen and Kowalski demonstrate how the latter requirement would contradict Goedel's incompleteness theorem; representability is thus a way of avoiding an overstatement of the logical power of this approach to metalanguage axiomatization.[16]

DLOG queries are alleged theorems of a DLOG database. The *query* predicate is true of all derivable theorems:

[16] See the discussion of provability in Bowen and Kowalski [Bowen82].

$$\forall x.derivable(DB,x) \supset query(x) \tag{3.25}$$

In other words, a successful evaluation of x as a query will result from demonstrating that x follows from database DB. Sentence (3.25) is a specification of the intended relationship between queries and databases in the DLOG data model. Standard implementations of resolution theorem provers provide one kind of implementation of "derivable" (e.g., [Chang73, Loveland78, Robinson79]). DLOG's derivable, as constructed in Prolog, is an incomplete but efficient implementation.

The distinction between constraints and implications is inherent in the relationship between DLOG databases and new assertions. This relationship is given by the following axiom:

$$\begin{aligned} &\forall x \forall X.assertion(x) \\ &\quad \wedge \forall y[y \in X \wedge constraint(y) \wedge relevant(x,y) \supset derivable(X \cup \{x\},y)] \\ &\quad \supset assertable(x,X,X \cup \{x\})) \end{aligned} \tag{3.26}$$

Intuitively, an assertion x can be added to a database X resulting in a new database $X \cup \{x\}$, if the relevant constraints of X (i.e., a subset of X) remain derivable when x is assumed as an axiom. "Relevant" is intended to mean "any constraint that might potentially be violated by the new assertion." Note that all constraints may be considered relevant; this would relieve all responsibility for specifying the meaning of "relevance," without affecting the DLOG specification. In this case, the definition of "relevant" is an efficiency issue; §5.2.4.2 provides an example specification.

The role and interpretation of integrity constraints has been discussed in the logic database literature from at least two viewpoints, one in which a database is viewed as a model of a set of constraints [Nicolas78b], and another in which integrity constraints are a distinguished subset of axioms of a first order theory which remain consistent over theory (i.e., database) updates [Reiter81, Kowalski81, Reiter83]. The former view, that of a database as a model of a set of constraints, is an accurate portrayal of traditional relational database systems. However the latter view is adopted here, where a database is viewed as a logical theory. The advantage is that integrity maintenance can be expressed in proof-theoretic terms: integrity constraints are those axioms that *should remain derivable* after a database update.

The proof-theoretic formalization of integrity constraints by Reiter [Reiter81, Reiter83] distinguishes a subset of axioms comprising a database $IC \subseteq DB$, and defines the notion of *constraint satisfaction*:

> A set of constraints IC is satisfied by a database DB if and only if,
> for all constraints $ic \in IC$, $DB \vdash ic$.

The definition simply says that the constraints IC are satisfied if and only if they can all be derived from the current database DB.

The formalization of Kowalski [Kowalski81] uses the same distinction, but does not characterize the class of constraints in enough detail to express an algorithm for their application — however, in Reiter [Reiter81], constraints are restricted to be of the form

$$\forall x_1 x_2 \cdots x_n . P(x_1, x_2, \ldots, x_n) \supset \tau^1(x_1) \wedge \tau^2(x_2) \wedge \cdots \wedge \tau^n(x_n)$$

where each τ^i is a boolean combination of monadic predicates. This restriction to monadic predicates is significant, as derivability of formulas of the monadic predicate calculus is decidable. In [Reiter83] this class is generalized to arbitrary formulas, but he acknowledges that their efficient application requires further research.

Two further observations on the use of integrity constraints are worth noting. First, constraints may sometimes be used to derive new information as well as to constrain the consistency of updates [Nicolas78b, Minker83]. For example, consider the formula

$$\forall x \forall y . supplies(x,y) \supset supplier(x) \wedge part(y) \qquad (3.27)$$

which asserts that if $supplies(x,y)$ is true, then x must be a supplier, and y a part. As an integrity constraint, formula (3.27) would cause the rejection of an assertion like "*supplies*(Acme,Widgets)" unless both "*supplier*(Acme)" and "*part*(Widgets)" were already derivable. However, as a ordinary rule we can use formula (3.27) to deduce that Acme and Widgets have the required properties. Similarly, in the terminal session in figure 2.5, the assertion

$$head(PCGilmore, CS)$$

was initially rejected because of the DDB constraint

$$\forall x \forall y . head(x,y) \supset faculty-member(x) \wedge department(y)$$

There seems to be no concensus about how to decide in which way to use formulas such as (3.26).

Second, because constraints are intended to assert invariant properties of databases, potentially violated constraints must be identified at update time. That is, upon each update, potentially violated constraints must be identified and re-derived to establish the consistency of the augmented database. Logic database researchers have usually distinguished integrity constraints from the rest of the database, but have not indicated how such a subset should be identified. This is presumably an issue of "database design."

In DLOG, constraints are intended to be used to specify invariant properties of those relations that a user may update. DLOG constraints are used only to verify database consistency, and cannot be used to answer queries. Furthermore, their syntax permits the definition of a meta predicate *relevant* that identifies a subset R of integrity constraints, i.e., $R \subseteq IC \subseteq DB$, that must be re-derived whenever a new assertion is made. This subset is determined by the form of the assertion (see §5.2.4.2).

The application of DLOG integrity constraints is specified in a modified version of Reiter's notion of constraint satisfaction [Reiter83]:

> if $DB \vdash ic$ for all $ic \in IC$,
> then A is assertable in DB only if
> $DB \cup \{A\} \vdash ic$ for all $ic \in IC$

If we assume that the current set of constraints IC is derivable, the consistent update of DB with new assertion A is possible only if $DB \cup \{A\} \vdash ic$, for all $ic \in IC$; the DLOG specification uses the meta predicate $assertable(A)$ as the specification of the DLOG mechanism for applying integrity constraints. Recall that the DLOG syntax for constraints has the form '$A \supset B$'; the DLOG database maintenance specification reveals it to be an abbreviation for '$assertable(A) \subset derivable(B)$'.

The class of constraints treated in [Reiter81] are equivalent to relational schemas, so DLOG's constraint facility also captures the database notion of relation schemas [Stonebraker75]. Furthermore, it is part of the data model — a constraint is simply another part of a DLOG application database.

Chapter 4
Descriptive terms in DLOG

Chapter 3 provided a rather detailed description of the DLOG data model, with little emphasis on the motivation for and intuition behind DLOG's complex terms. This chapter presumes to supply the motivation, with further examples of their intended use.

The motivation for including descriptive terms in DLOG comes from work in Artificial Intelligence, where the ability to refer to objects with descriptions is of considerable conceptual advantage in specifying symbolic representations of domains (e.g., [Moore76, Schubert76, Norman79, Bobrow77b, Bobrow77a, Hewitt80, Steels80, Attardi81]).

Descriptions have also been used within the logic database literature (e.g., [Dilger78, Dahl80, Dahl82]). The descriptive terms of DLOG are believed to be the most extensive currently employed in any logic-based representation language.

In AI, part of the reluctance to adopt a logical interpretation of descriptions must be attributed to the long-standing confusion and controversy about their meaning — logicians do not generally agree on the semantics of descriptive terms.[17] This reluctance to analyze descriptions in a logical framework probably results from a traditional misunderstanding of the role of logic (cf. [Hayes77]). DLOG descriptions were originally developed to help describe concepts in the Department Database (DDB) domain; they have been somewhat elaborated to help explain the role of logic in analyzing descriptions and to investigate the computational problems of including them in a representation language.

The next section reviews all DLOG complex terms, and discusses their use and intended meaning in an informal way. Two subsequent sections, §§4.2, 4.3, discuss the traditional use and meaning of descriptions by focusing on issues that impact their interpretation in a finite symbolic database. The

[17] See [Carroll78] for a good survey of the logical, philosophical and psychological interpretation of descriptions.

chapter's last section, §4.4, gives more examples of how DLOG descriptive terms are used in the DDB domain.

4.1. DLOG terms: motivation and intended use

When designing a representation language, it is easy to become detached from the worlds to be represented; elaborate language features are often presented with little indication of how they are best used. For example, the KRL language [Bobrow77b, Bobrow77a] has been criticised in this way (e.g., see [McDermott78b, Lehnert78]).

The need for embedded descriptions does arise in the DDB domain (e.g., see fig. 4.1 below). In DLOG's case, indefinite descriptions, lambda terms, and indefinite sets were motivated by domain considerations. Definite descriptions of both kinds were used rarely in the implemented DDB, but were included in order to investigate strategies for their implementation.

4.1.1. Individual constants

DLOG individual constants are, by Tarskian semantics, intended to denote individuals in an interpretation. A constant identifier is usually selected as a name of an individual, but this need not be the case. The correspondence between constants and real objects is a semantic issue, so a constant like *PERSON*00169 might be used to denote the person named John Q. Smith. Conceptually, it is generally less confusing to choose identifiers mnemonically. An alternative is to specify a name relation as a predicate of the theory, e.g., establish a correspondence like

name(*P*000169, *"John Q. Smith"*)
name(*P*000072, *"Dweisel Zappa"*)
.
.
.

The non-mnemonic constants might then be interpreted as denoting individuals in the domain, with the predicate *name* interpreted as a relation on individuals and strings. In this way, a user can build a database that includes a theory of aliases, e.g.,

name(*P*000185, *"Clark Kent"*)
name(*P*000185, *"Superman"*)

4.1.2. Set constants

DLOG set constants are formed from individual constants, so each member of a set is interpreted as specified above for individual constants. A set constant is written as a collection of individual constants, e.g., $\{P000119 \wedge P10112\}$, and denotes a collection of individuals. For example, the axiom $\Phi(\{P000119 \wedge P10112\})$ is true in an interpretation where the property named by Φ is true of the class object that has the denotations of *P*000119 and *P*10112 as members.

Note that a DLOG theory is not a complete axiomatization of classical set theory (cf. [Brown78]). As Quine says:

> ...set theory can in part be simulated by purely notational convention, so that the appearance of talking of sets (or classes), and the utility of talking of them, are to some degree enjoyed without really talking of anything of the kind.[18]

Similar observations have been made by Bledsoe [Bledsoe77a], who replaces set variables with their first order translations, and Kowalski [Kowalski81], who uses a similar idea to provide an axiomatization of sets.

DLOG's set theory is based on a finite domain of individuals, and thus avoids the major issue of classical set theory: what expressions determine sets, and vice versa. This is done by restricting the interpretation of sets to range over the power set of a finite number of individuals. A logician would no doubt claim that, indeed, DLOG has sets "without really talking of anything of the kind." However, the intended use of DLOG "sets" is as a conceptual aid — they behave as if they were sets.

Recall that chapter 3 presented the DLOG language as a 3-sorted logic. DLOG's interpretation of set constants could be transformed to a single-sorted language. This translation to a single-sorted logic requires that each application database include axioms that define the unary predicates *individual*, *set* and λ. In fact, because the implementation of DLOG is in the single-sorted language

[18] See [Quine69, p. 4, pp. 15-21].

Prolog, the unary predicates are required for the implementation, and are provided for the user. DLOG users can assume that the necessary axioms are "built-in," however an assertion of the form $\Phi(\{P000119 \wedge P101112\})$ is manipulated in a single-sorted language internally. By incorporating these predicates within each DLOG database, the user is provided with the illusion of sets.

4.1.3. Lambda constants

The experimental DDB domain requires the description of degree requirements. Often the requirements can be formed as lambda constants, e.g., the assertion

$$enrolment-requirement(BScMajorsCS, \lambda x.[age-of(x,y) \wedge y \geq 16]) \tag{4.3}$$

states that "an enrolment requirement of the BScMajorsCS degree is that the candidate's age is greater than or equal to sixteen." (Recall that all unbound variables in the body of a lambda term are assumed to be existentially quantified.) The lambda constant format allows the requirement to be asserted and queried, and the *satisfies* predicate provides the mechanism to pose a query like

$$?enrolment-requirement(BScMajorsCS, l_1) \wedge satisfies(John, l_1) \tag{4.4}$$

that can be read as "Has John satisfied an enrolment requirement for the BScMajorsCS program?"

Notice that, in the DDB domain, degree requirements are most naturally conceived as conditions which must be satisfied. Since degree programs are distinguished by their various requirements, it is most straightforward to describe degree program requirements as relations on degree names and conditions to be satisfied — in DLOG, as lambda constants.

Of course there are alternatives to the use of this special constant. For example, the meaning of sentence (4.3) might be rephrased in terms of a standard first order language as

$$\forall x.[satisfied-requirements(x, BScMajorsCS) \supset \exists y.age-of(x,y) \wedge y \geq 16] \tag{4.5}$$

where we would use *BScMajorsCS* as the name of a degree program and modify the predicate *satisfies* to correspond more closely to our intuition regarding what one must do with degree requirements. This alternative has a more straightforward meaning (since there are no "special" forms) and the requirement for a higher order semantics is gone. But now there is no way of asking what the requirements of the BScMajorsCS program are, short of providing another meta level order primitive for manipulating sentences. For example, to answer the query equivalent to the query (4.4) in the alternative notation, we require an operation that retrieves a sentence of the form (4.5) from the current database, and then returns the literal consequent of that sentence as an answer.

Similarly, if there were n lambda constants, each denoting a degree requirement regulation, .i.e.,

$$\begin{array}{l} requirement(BScMajorsCS, \lambda x.\Phi_1) \\ requirement(BScMajorsCS, \lambda x.\Phi_2) \\ \cdot \\ \cdot \\ \cdot \\ requirement(BScMajorsCS, \lambda x.\Phi_n) \end{array}$$

we could rewrite this collection of atomic formulas as

$$\forall x \Phi_1 \wedge \Phi_2 \wedge \cdots \Phi_n \supset satisfied-requirements(x, BScMajorsCS)$$

However, as above, any attempt to ask a questions about degree requirements would again require meta level manipulation of the second formula.

Lambda terms can be manipulated with a standard (sorted) proof procedure to answer existential queries about requirements; they are simply retrieved and bound to existential lambda (predicate) variables as in normal answer extraction. Furthermore, they can be used in conjunction with the *satisfies* predicate to determine if an individual has satisfied a particular requirement.

4.2. Problems with interpreting descriptions

The major conceptual advantage of descriptions is their ability to refer to an object by specifying its properties, rather than by specifying a unique name. Confusion about their intended meaning arises when they fail to refer, or when multiple referents result in ambiguous reference.

The most common interpretation of descriptive terms is provided by Russell's method of contextual definition (e.g., see [Kaplan75]). Russell proposed that sentences of the form

$$bald(\iota x_1.king-of-France(x_1)) \tag{4.6}$$

should be viewed as an abbreviation for

$$\exists x_1.[bald(x_1) \wedge \forall x_2.[king-of-France(x_2) \equiv x_2 = x_1]] \tag{4.7}$$

At issue is the meaning of descriptive terms when their proof-theoretic preconditions fail.[19] For example, Russell held that if there were no such King of France (as apparently referred to in sentence 4.6), then the sentence was false. His claim rested on the truth value of sentence (4.7) which is the definition of the abbreviated sentence (4.6). Still another popular theory[20] holds that the meaning of such descriptions whose logical preconditions fail should be specified by convention, e.g., a failing description refers to a designated null constant that lies outside the domain of discourse.

Logicians have not been primarily concerned with the conceptual advantage of descriptions (e.g., [Hilbert39, Leisenring69, Robinson79]), and have thus been content with rather intuitive interpretations so long as they remain within proof-theoretic constraints (e.g., see [Rosser78]). However, the use of descriptions in AI and logic databases is primarily of conceptual advantage, and therefore requires care in specifying their meaning with respect to the languages in which they appear (e.g., see [Moore76, Schubert76, Dilger78]). In this regard, the use of descriptive terms impinges on philosophical problems associated with names and reference (e.g., [Donnellan66, Kaplan75, Brinton77, Katz77]). Here the distinction between *referential* and *attributive* use is an important concept in understanding the use of descriptions [Donnellan66]. This distinction has received scant attention in AI, except in one semantic network representation [Schubert76] and in a cognitive theory of description use [Ortony77]. Briefly, the distinction hinges on the description's *intended* use: if a description is formed with the intention of referring to a particular individual, the use is referential; if a description serves only to identify the properties of some referent without intentional concern for its existence, the use is attributive. Examples from Anderson and Ortony [Ortony77] will illustrate: compare

The inventor of dynamite wore a fine beard. (4.8)

with

[19] E.g., see [Carnap47, pp. 32-39].

[20] See discussion of the Frege-Carnap theory in [Kaplan75], page 215.

The inventor of dynamite had a profound influence (4.9)
on the nature of warfare.

The former sentence seems to be referring to a particular individual (i.e., Alfred Nobel), while the latter is only coincidentally concerned with the referent of "the inventor of dynamite."

In *any* knowledge base, the user's intention must be communicated to the system, or the system must impose some decision regarding referential versus attributive usage. As Carroll states:

> Analyses that reject this distinction or lack the formal apparatus to describe it, just cannot treat everyday facts about naming and reference.[21]

If we insist that descriptive terms denote, we can retain our logical analysis (i.e., contextual definition) by arbitrarily forcing every use to have a corresponding referent. When referential use fails, a new object is created and the description is interpreted as a reference to some existing but undistinguished individual. This is the approach suggested by Schubert [Schubert76].

In logic databases, the unnamed object created as a result of the attributive interpretation is a new Skolem constant [Reiter83]. For example, when interpreting the description (4.8), failure to find an existing constant that satisfies the property "the inventor of dynamite" might sanction the creation of a new object, asserted to be the one in question. Furthermore, it is asserted to have the property of "wearing a fine beard." Each such attributive interpretation will require appropriate changes to the logic database axioms for equality (see §3.4.2), as the new object must be included in those known to exist, and the extensions of relevant predicates must be updated.

Note that the inequalities that record the unique name assumption are not changed. A newly created Skolem constant cannot be assumed to be distinct from some existing constant. In our previous example, a user's attributive use of "the inventor of dynamite" may be made in ignorance of some existing constant denoting the individual Alfred Nobel. The subsequent problem of re-establishing name uniqueness after each attributive use is computationally volatile — it requires an equality test with each existing constant..

---

[21] See [Carroll78, p. 29].

As regards assertions, we know of no logic database system that allows descriptive terms. For DLOG, the suggestion of Schubert's is applicable, but some decisions remain unclear. For example, an attributive treatment of definite descriptions might use the description itself as the Skolem constant, or create a new constant and discard the description, or create the new constant and save the description. For instance, consider the DLOG equivalent of (4.8):

$$wore-a-fine-beard(\iota x.inventor-of-dynamite(x)) \qquad (4.10)$$

If a referent is not found, should (4.10) be asserted? Or, should something like

$$wore-a-fine-beard(P010174)$$

be asserted? Should the equivalence

$$\iota x.inventor-of-dynamite(x)=P010174 \qquad (4.11)$$

be asserted as well, or how about

$$inventor-of-dynamite(P010174)?$$

For definite descriptions, asserting anything equivalent to (4.11) means that every subsequently created individual constant must be tested for identity with the Skolem constant to ensure database integrity. This might be avoided by throwing away all descriptions, but some recent work on intelligent interfaces indicates that descriptions are extremely useful as query responses [Kaplan79, Kaplan82]. In fact it is more difficult to recreate a description than to retain the description when it is asserted.

The interpretation of descriptions is further complicated when the embedding language is used to form queries as well as assertions, as in some languages based on logic [Dilger78, Dahl80]. It seems intuitively natural to treat descriptions referentially in queries, since we are presumably forming a query to identify objects and relationships. The lambda descriptions of Dilger and Zifonun [Dilger78] are used in exactly this fashion. Furthermore, in addition to the conceptual advantage of using descriptions referentially, Dahl [Dahl80] notes that referential interpretation of set descriptions can provide a solution to the problems arising in the interpretation of negation in Prolog. Briefly, the problem is in computing the complement of a predicate extension when deriving a negated literal of that predicate. For example, in Prolog a derivation of the query $\exists x.\neg P(x)$ from $DB=\{P(a),Q(b)\}$

fails, even though $x = b$ is a legitimate answer. This is because Prolog's negation-as-failure mechanism [Clark78] attempts to show $DB \vdash \exists x \neg P(x)$ by instead showing $DB \nvdash \exists x P(x)$; here, since $DB \vdash P(a)$, the negated query fails.

Dahl's suggests two ways to ensure that a query of the form $\exists x \neg P(x)$ is never attempted; in its place she puts one of two mechanisms that can identify the members of the extension of predicate P, and then test each individually. A different heuristic is used in the MU-Prolog system [Naish83a]. MU-Prolog will delay the execution of a negated atom as long as possible, hoping that free variables in the negated atom will become bound; negation-as-failure works properly as long as the x in '$\exists x \neg P(x)$' is bound. This solution is possible only because of the finite axiomatizations being considered.

4.3. Interpreting DLOG descriptions

The DLOG data model allows descriptions to appear in both assertions and queries (cf. [Dilger78]). Here we provide an intuitive specification of descriptions, as an adjunct to §3.2.1.

4.3.1. Definite individuals

Recall (§3.1.1, syntax description T3) that a definite description of an individual in DLOG is called a definite individual, and is written using the iota variable binding operator, e.g.,

$$\iota x_1.inventor-of-dynamite(x_1)$$

The DLOG data model treats these descriptions referentially in both assertions and queries. At assertion time, a definite individual that fails to refer should cause rejection of the assertion. This is merely a part of the DLOG specification, and no implementation commitment is intended. The specification could be made more precise by using the same technique used to specify DLOG database maintenance (§3.6). For example, for every use of a definite individual $\Phi(\iota x_1.\Psi(x_1))$, a meta axiom schema instance like

$$assertable(\Phi(\iota x_1.\Psi(x_1)),DB,DB \cup \{\Phi(\iota x_1.\Psi(x_1))\}) \subset$$
$$derivable(DB,\forall x_2 x_3.\Psi(x_2)) \wedge \Psi(x_3)) \supset \wedge x_2 = x_3.$$

must be considered. As queries are viewed as alleged theorems, a definite individual that fails to refer results in query failure. This interpretation of definite individuals is essentially the test of an assertion's presuppositions regarding the individual being described (cf. [Mercer84, Hirschberg84]). Here the presupposition is existence and uniqueness.

4.3.2. Indefinite Individuals

Indefinite descriptions of individuals are formed using Hilbert's symbol ϵ, intuitively interpreted as the English indefinite article "an." It should here be acknowledged that, whereas philosophical treatments of reference have been most interested in the meaning of definite descriptions, theories of indefinite descriptions have received relatively less attention. For example, Hilbert's operator can be used to dispense with quantifiers but it does so only for syntactic expedience, to ease the mechanical function of the proof theory [Leisenring69]. It is apparently for this reason that Kaplan [Kaplan75, p. 213] denies Hilbert's term the status of an indefinite description operator even though there is some basis to believe that Hilbert intended its interpretation as a designator could be based on a semantic choice function (e.g., see [Leisenring69, Alps81]).

In DLOG, the term

$$\epsilon x.student(x) \wedge age-of(x,16)$$

can be read as "a student whose age is 16." Indefinite descriptions are treated, in a sense, attributively in assertions, and referentially in queries. In assertions they are interpreted attributively in the sense that that failure of their preconditions (i.e., existence) cannot cause an assertion to be revoked (cf. definite descriptions). An assertion of the form

$$\Phi(\epsilon x.\Psi(x))$$

will be admitted as an axiom regardless of whether the clause $\Psi(x)$ is derivable (i.e., there is a referent for x). This means that an indefinite assertion of the form

$$\Phi(a) \vee \Phi(b)$$

can be made by embedding the disjunction within a description, viz.

$$\Phi(\epsilon x.x=a \vee x=b)$$

This ability to express incomplete information has been claimed as a major advantage of logic database research over traditional relational databases (e.g., [Reiter83, Minker83]). However, the current

DLOG implementation is based on a Horn clause theorem prover that cannot derive disjunctive theorems. Minker's recent proposal [Minker83] for a method of evaluating queries against an indefinite database shows that a Horn clause theorem prover could be augmented to deal with these indefinite assertions; §5.4 discusses a heuristic method for manipulating indefinite assertions, and suggests how it can be used to retrieve indefinite information even when that information is not retrievable by the standard answer extraction method.

In queries, indefinite descriptions are treated referentially, as they would be in their standard rendering. In other words, indefinite descriptions encountered during the pursuit of a proof are subject to reference determination. This treatment of indefinite individual descriptions is practical because, in contrast to their definite counterparts, no uniqueness property must be maintained.

4.3.3. Definite sets

The expression

$$\{x_1{:}\Phi(x_1)\}$$

is interpreted as a reference to the set of all individuals x_1, such that $\Phi(x_1)$ is derivable. No evaluation of a definite set description is made at assertion time, but its treatment can be considered as referential. As mentioned in §3.1.1, the description is treated as *definite* because it refers to the set of *all* appropriate individuals at any time during the life of the database. When interpreted in a query, the definite set description refers to the current extension of the described set — it is (trivially) unique.

4.3.4. Indefinite sets

Definite and indefinite sets are distinguished by the uniqueness of the former, just as for definite and indefinite individuals. An indefinite set is specified by a description of a typical element (as for definite sets), but in addition, by a predicate that may be derivable for several such sets. For example, consider an example as specified by syntax description (T8.1), §3.1.1:

$$\{x_1,X_1{:}course(x_1)\wedge cardinality-of(X_1,3)\}$$

The first conjunct of the description describes a typical member of the set, in this case '$course(x_1)$';

the set to which this description refers will be an element of the subsets of the set of all courses. The second conjunct names a property of sets rather than individuals, viz. '$cardinality-of(X_1,3)$'; the subsets of the "typical member set" with this property are the ones that satisfy the body of the set description.

The restrictive interpretation of DLOG theories provides a computational view. The two parts of the description body can be viewed as follows: the typical member description is a "generator," and the set property is a "terminator." (The notion of generator is the same as that in Turner's KRC language [Turner82]). To compute the family of sets referred to, we use the generator to construct the power set of typical elements in some order, then apply the terminator to each one. Each member of the powerset that satisfies the terminator is in the family of sets; the description refers to one of that family.

This indefinite description mechanism can be used for writing numerical quantifiers [Brachman78, Shapiro79], simply by writing a cardinality restriction as a terminator. Furthermore, it subsumes the quantification scheme of the KS system [Dilger78] because *any* set relation can be used to form the indefinite set property.

The motivation for indefinite sets derives from the Department Database domain, and assertions like "At least 12 CS courses are required for the BSc degree in CS." The initial rendering of this statement might be as something like

$$\left\{\begin{array}{l} course-req(BSc-CS,CS115) \\ \wedge course-req(BSc-CS,CS215) \\ \wedge \\ \cdot \\ \cdot \\ \cdot \\ \wedge course-req(BSc-CS,CS442) \end{array}\right. \tag{4.12}$$

$$\vee \left\{\begin{array}{l} course-req(BSc-CS,CS118) \\ \wedge course-req(BSc-CS,CS215) \\ \wedge \\ \cdot \\ \cdot \\ \cdot \\ \wedge course-req(BSc-CS,CS442) \end{array}\right.$$

$$\begin{array}{c} \cdot \\ \cdot \\ \cdot \end{array}$$

where the components grouped by the left brace must be written once for each set of twelve CS courses. Note that we are trying to describe the set of all sets consisting of twelve CS courses. This is clearly a tiresome way to express the assertion, and furthermore, would require extensive modification after any new CS course was created (e.g., by adding an assertion like "course(CS123)").

In general, we can reduce the fatigue of writing such assertions by noting that assertions of the form "At least n with property P" can be expressed as

$$\begin{array}{l} \exists x_1 x_2 \cdots x_n .[x_1 \neq x_2 \wedge x_1 \neq x_3 \cdots x_{n-1} \neq x_n \\ \qquad \wedge P(x_1) \wedge P(x_2) \cdots \wedge P(x_n)] \end{array} \tag{4.13}$$

where the first portion of the assertion expresses that each such object is unique and the second portion attributes the property P to each object identified. We can then make the desired reference to all sets with property P by introducing set variables and writing the formula

$$\exists x_1.[\forall x_2.[x_2 \in X_1 \supset P(x_2)] \wedge cardinality-of(X_1,x_1) \wedge x_1 \geq n] \tag{4.14}$$

Notice that the set variable 'X_1' is free — we are writing a formula that describes all such sets.

For the next elaboration, we choose some variable binding operator, say 'δ', to mean "one of"; we then substitute '12' for 'n' and 'topic-of' for 'P' in (4.14) to get

$$\delta X_1.\exists x_1.[\forall x_2.[x_2 \in X_1 \supset topic-of(x_2,CS)] \wedge cardinality-of(X_1,x_1) \wedge x_1 \geq 12] \tag{4.15}$$

Expression (4.15) is then embedded within the *course−req* predicate to form the assertion

$$\begin{aligned}&course-req(BSc-CS,\\ &\delta X_1.[\forall x_1.[x_1 \in X_1 \supset topic-of(x_1,CS)] \wedge \exists x_2.cardinality-of(X_1,x_2) \wedge x_2 \geq 12])\end{aligned} \tag{4.16}$$

Notice that the description (4.15) is an instance of the contextual definition of indefinite sets described in §3.2.1 (equation 3.4). The description body describes a typical set element "CS course" and the property of the set in question "at least twelve members." In the DLOG syntax we can write (4.16) as

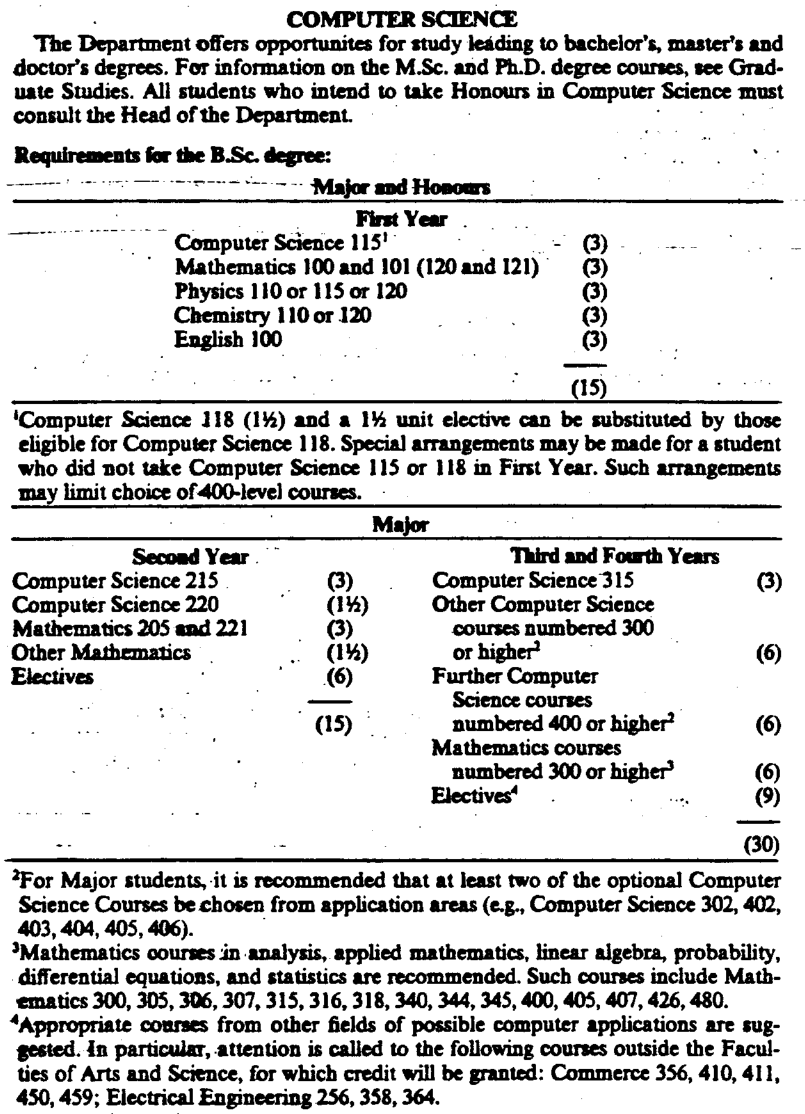
$$\begin{aligned}&course-req(BSc-CS,\\ &\qquad \{x_1,X_1:topic-of(x_1,CS) \wedge (cardinality-of(X_1,x_2) \wedge x_2 \geq 12)\})\end{aligned} \tag{4.17}$$

In the intended interpretation some ambiguity remains, as the original assertion (4.12) is a relation on individuals (e.g., degree names × courses) but the final version (4.17) involves sets (e.g., degree names × sets of courses). Intuitively, we expect that if a set of courses is a course requirement, then so is each member of that set. The DLOG implementation described in chapter 5 uses a weak typing mechanism that permits the user to declare how the set arguments of each predicate are to be interpreted.

4.4. Mapping a domain into DLOG sentences

The specification of the DLOG data model provides a set of guidelines for implementing a DLOG database management system, as well as a description of how to capture and interpret a domain in terms of DLOG sentences. The claimed advantage of including the numerous kinds of special terms is conceptual ease in mapping a domain to DLOG sentences. Much of the difficulty in using a representation language or data model is deciding how to capture relevant domain facts in a database. This section provides some examples of the intuition behind one particular rendering of the example DDB domain.

Consider the information in fig. 4.1, which is a reproduction of a portion of The University of British Columbia undergraduate calendar for the academic year 1981-1982.



COMPUTER SCIENCE

The Department offers opportunites for study leading to bachelor's, master's and doctor's degrees. For information on the M.Sc. and Ph.D. degree courses, see Graduate Studies. All students who intend to take Honours in Computer Science must consult the Head of the Department.

Requirements for the B.Sc. degree:

Major and Honours

First Year	
Computer Science 115[1]	(3)
Mathematics 100 and 101 (120 and 121)	(3)
Physics 110 or 115 or 120	(3)
Chemistry 110 or 120	(3)
English 100	(3)
	(15)

[1]Computer Science 118 (1½) and a 1½ unit elective can be substituted by those eligible for Computer Science 118. Special arrangements may be made for a student who did not take Computer Science 115 or 118 in First Year. Such arrangements may limit choice of 400-level courses.

Major

Second Year		**Third and Fourth Years**	
Computer Science 215	(3)	Computer Science 315	(3)
Computer Science 220	(1½)	Other Computer Science courses numbered 300 or higher[2]	(6)
Mathematics 205 and 221	(3)	Further Computer Science courses numbered 400 or higher[2]	(6)
Other Mathematics	(1½)	Mathematics courses numbered 300 or higher[3]	(6)
Electives	(6)	Electives[4]	(9)
	(15)		(30)

[2]For Major students, it is recommended that at least two of the optional Computer Science Courses be chosen from application areas (e.g., Computer Science 302, 402, 403, 404, 405, 406).

[3]Mathematics courses in analysis, applied mathematics, linear algebra, probability, differential equations, and statistics are recommended. Such courses include Mathematics 300, 305, 306, 307, 315, 316, 318, 340, 344, 345, 400, 405, 407, 426, 480.

[4]Appropriate courses from other fields of possible computer applications are suggested. In particular, attention is called to the following courses outside the Faculties of Arts and Science, for which credit will be granted: Commerce 356, 410, 411, 450, 459; Electrical Engineering 256, 358, 364.

Figure 4.1 A portion of the DDB domain

As described in §2.3, the experimental DDB is intended for use in building and maintaining student transcripts. The question answering knowledge is informally classified as knowledge of program requirements and knowledge of program prerequisites. Program requirements include faculty program requirements, department program requirements, and course requirements, e.g.,

$$\begin{aligned} & faculty-program-req(Science,\\ & \quad Bachelor,\\ & \quad x_1,\\ & \quad second,\\ & \quad \lambda x_2.completed(x_2,\\ & \qquad \{x_3,X_1:(\ course(x_3)\\ & \qquad\quad \wedge units(x_3,x_4)\ )\\ & \qquad \wedge(\ total-units(X_1,x_5)\\ & \qquad \wedge x_5 \geq 24)\ )\}\ ) \end{aligned} \tag{4.18}$$

In general, $faculty-program-req(\alpha,\beta,\gamma,\delta,\epsilon)$ is interpreted as "in faculty α, at level β, in stream γ and year δ, requirement ϵ must be satisfied." Assertion (4.18) specifies a second year requirement for Bachelor of Science programs, for all streams x_1 (e.g., Majors and Honours). The requirement is that a student x_2 have completed a set of courses, specified as an indefinite set description; the indefinite set is read as "a set of courses whose total number of course units is greater than or equal to twenty-four." Notice that the *total−units* predicate is applied to set terms; it can be defined in terms of the simpler *units* predicate. We expect to be able to distinguish how a predicate is being used by determining the sorts of its arguments. One implementation technique for doing so is described in chapter 5.

Requirements are represented by lambda terms because, in the domain, they are perceived as regulations, or truth conditions to be satisfied. As explained above, in §4.1.2, queries can retrieve requirements as *terms*, and use the definition of *satisfies* to determine if some particular individual satisfies that requirement.

Faculty program requirements are related to program requirements by the implication

$$\begin{aligned} & program-req(x_6,x_4,x_5) \subset\\ & \qquad faculty-program-req(x_1,x_2,x_3,x_4,x_5) \end{aligned}$$

$$\begin{aligned}&\wedge faculty-of(x_6,x_1)\\&\wedge level-of(x_6,x_2)\\&\wedge stream-of(x_6,x_3)\end{aligned}$$

In a similar way, department and course requirements are included in the class of program requirements.

Most of the partial domain given in fig. 4.1 specifies course requirements. Consider the first course requirement for first year B.Sc. Majors:

$$\begin{aligned}&course-req(BScMajorsCS,\\&\quad first,\\&\quad \lambda x_1.completed(x_1,CS115)\\&\quad\quad \vee(completed(x_1,CS118)\\&\quad\quad\quad \wedge completed(x_1,\epsilon x_2.elective(BScMajorsCS,x_2)\\&\quad\quad\quad\quad\quad\quad \wedge units(x_2,1.5)))\end{aligned}$$

Notice that this requirement makes use of an indefinite description of an individual in rendering the domain requirement "a 1½ unit elective." An alternative representation might have been

$$\begin{aligned}&course-req(BScMajorsCS,first,CS115)\\&\quad\quad \wedge\ completed(x_1,CS115)\\&\quad\quad\quad \vee\ (course-req(BScMajorsCS,first,CS118)\\&\quad\quad\quad\quad \wedge completed(x_1,CS118)\\&\quad\quad\quad\quad \wedge elective(BScMajorCS,x_2)\\&\quad\quad\quad\quad \wedge units(x_2,1.5)\\&\quad\quad\quad\quad \wedge course-req(BScMajorsCS,x_2)\\&\quad\quad\quad\quad \wedge completed(x_1,x_2))\end{aligned}$$

The descriptive version shortens the representation by keeping the alternative course within the context of *course−req*. Furthermore, the lambda constant allows the requirement to be tested for satisfaction.

As a final example, notice that the third and fourth year requirements for the Majors program are specified together in fig. 4.1. Consider the following as a representation of "6 units of other Computer Science courses numbered 300 or higher:"

$$\begin{aligned}
&course-req(BScMajorsCS,\\
&\quad \{third \wedge fourth\},\\
&\quad \lambda x_1.completed(x_1,\\
&\qquad \{x_2,X_1:(course(x_2) \wedge dept-of(x_2,CS)\\
&\qquad\quad \wedge course-no(x_2,x_4)\\
&\qquad\quad \wedge x_4 \geq 300)\\
&\qquad \wedge units(X_1,6)\}))
\end{aligned}$$

Since this requirement is specified for third and fourth year, the assertion is made for the set term that contains *third* and *fourth* as individuals. Notice that the set term $\{third \wedge fourth\}$ means that the assertion is not decomposable into two assertions of the form $course-req(BScMajorsCS,third,—)$ and $course-req(BScMajorsCS,fourth,—)$ because the lambda constant is a requirement that must be satisfied in the third and fourth year period. The intended interpretation of the predicate *course−req* is such that both individual and set arguments in the second position make sense. In the implementation of chapter 5, a user is given some control over the specification of predicate constants, to achieve this flexibility in using sorts. The intuitive meaning of the above combination of predicate constants implicitly asserts something like

$$\forall x_1 x_2 x_3 X_1.course-req(x_1,X_1,x_2) \wedge x_3 \in X_1 \supset course-req(x_1,x_3,x_2)$$

This kind of flexibility is provided by the implementation.

Footnote "2" in this specification (see fig. 4.1) describes recommendations; in general recommendations must be related to indefinitely specified requirements by some user decision, but the recommendations themselves can be captured in the sentence

$$\begin{aligned}
&recommended(BScMajorsCS,\\
&\quad \{third \wedge fourth\},\\
&\quad \{x_1,X_1:(course(x_1)\\
&\qquad \wedge(x_1 \in \{CS302 \wedge CS402 \wedge CS403 \wedge CS404 \wedge CS405 \wedge CS406\})\\
&\qquad \vee(topic-of(x_1,applications))\\
&\quad \wedge(cardinality(X_1,x_2) \wedge x_2 \geq 2)\})
\end{aligned}$$

that recommends "at least two" of those suggested courses.

Chapter 5
An implementation of the DLOG data model

An experimental database management system based on the DLOG data model has been implemented in the programming language Prolog. Two different implementations have been developed, one in Waterloo IBM Prolog under the CMS operating system [Roberts77], and another in Prolog/MTS [Goebel80], a derivative of Waterloo IBM Prolog. Another version is currently under construction in Waterloo Unix Prolog [Cheng84].

A Prolog interpreter is a flexible implementation tool because it provides a theorem-prover, a database management system, and a programming language all in one package. This accounts for its growing popularity in many areas of computing (e.g., see [Clark82]).

5.1. Prolog overview

This overview is necessarily brief, but should provide the concepts needed to interpret the specification of DLOG in Prolog. For a programming introduction, see Clark and McCabe [Clark84].

The syntax we will use is that of Prolog/MTS. The language is single sorted. Variables are written as arbitrary alphanumeric strings preceded by asterisks, e.g.,

$$human(*x)$$

is Prolog's representation for

$$\forall x.human(x)$$

Implications are written in the form

$$consequent \ \texttt{<-} \ antecedent$$

and are interpreted by Prolog to mean "to establish *consequent* try deriving *antecedent*." For example, the axiom

$mortal(*x) <\!\!- human(*x)$

is Prolog's representation for

$\forall x.human(x) \supset mortal(x)$

To prove that some α is mortal, one can show that α is human. Prolog queries or goal statements are conventionally written as

$<\!\!- goal_1 \& goal_2 \& \cdots \& goal_n.$

If the free variables are $x_1 x_2 \cdots x_m$, this goal is considered as a request to establish

$\exists x_1 x_2 \cdots x_m .goal_1 \wedge goal_2 \wedge \cdots \wedge goal_n$

as a theorem; all free variables are interpreted as existential. The back-chaining (i.e., goal-driven) proof procedure is a depth-first implementation of the SLD proof procedure [Lloyd82]. Finite failure on one search path will force backtracking to another path; terms are substituted for existential variables where appropriate.

The meta variable facility of Prolog is an implementation of the reflection rule used by Weyrauch [Weyrauch80] and Bowen and Kowalski [Bowen82]. It is simply a way of treating the value of a bound variable as a goal, e.g.,

$derivable(*x) <\!\!- *x$

is a Prolog statement representing the meta language statement

if $DB \vdash x$ then $DB \vdash derivable(x)$

Intuitively, this axiom states that an attempt to establish x as a theorem using *derivable*, is equivalent to deriving x directly in the logic's proof procedure. The axiom's validity is established when the proof procedure axiomatized by *derivable* corresponds to the logic's proof theory (i.e., the definition of '$\vdash$'). This rule permits the logic programmer to either simulate a proof procedure using *derivable*, or to relinquish control to the Prolog proof algorithm and pursue a proof directly. By simulating a derivation, a programmer can control each derivation step and substitute alternative proof strategies or apply heuristics: this is how the DLOG implementation modifies Prolog's standard proof theory to deal with descriptions. Inefficiency is the penalty for simulating derivations: a derivation of $derivable(x)$ is simply a direct proof of x.

By using the meta variable mechanism, the Prolog implementation of DLOG can manipulate DLOG sentences as terms, and can use Prolog's proof theory to derive sentences that fall within the standard Prolog interpretation. This is the mechanism required to manipulate DLOG higher-order objects.

5.1.1. Prolog as a foundation for DLOG

The DLOG data model is an extension of the data model implicit in Prolog [Kowalski81]. Prolog's data model uses the Horn clause subset of a first order language as its representation language, Tarskian semantics, and a depth-first proof procedure based on SLD resolution. Prolog's treatment of negation is defined by Clark's results [Clark78].

From this viewpoint, DLOG extensions include complex terms and constraints. These extensions require features unavailable with the implicit data model of Prolog, but they are provided by treating the Prolog language as its own meta language in which the appropriate extensions can be axiomatized [Bowen82].

The DLOG extensions are defined, indeed *implemented*, by using Prolog to describe their relationship to Prolog's implicit data model. The handling of complex terms is provided by extending Prolog's theory of equality, as implemented in unification; the interpretation of lambda constants and constraints is specified by a meta language axiomatization. For example, the specification of DLOG database maintenance requires that we implement axiom (3.26) of §3.6

$$\begin{aligned}&\forall x\,\forall X.assertion(x)\\&\qquad\wedge\forall y[y\in X\wedge constraint(y)\wedge relevant(x,y)\supset derivable(X\cup\{x\},y)]\\&\qquad\supset assertable(x,X,X\cup\{x\}))\end{aligned}\tag{5.1}$$

We can massage formula (5.1) into a Prolog-like syntax by first rewriting it as:

$$\begin{aligned}assertable(x,X,X\cup\{x\})\subset\ &assertion(x)\\&\quad\wedge\forall y.y\in X\\&\qquad\wedge constraint(y)\end{aligned}\tag{5.2}$$

$$\wedge relevant(x,y) \supset derivable(X \cup \{x\},y)$$

Here we have reversed the implication sign, then dropped the universal quantifiers on x and X because free variables in the consequent of an implication are interpreted by Prolog as universal throughout the formula. However, we have explicitly retained the universal quantifier within the antecedent of formula (5.2), as Prolog will interpret variables appearing only in the antecedent of an implication as existential.

So far, formula (5.2) is similar to the database maintenance axioms proposed by Bowen and Kowalski [Kowalski79, Bowen82], but notice that it is not a Horn clause (an explicit universal quantifier appears in the antecedent). Kowalski's more recent formulation does not have that problem:[22]

$$\begin{array}{l} Assimilate(currdb, constraints, input, bound, newdb) \\ \quad \textbf{if } Demo(currdb \cup constraints, bound, not(input), yes(proof)) \\ \quad \textbf{and } AnalyseFailureRestoreConsistency(currdb, constraints, proof, newdb) \end{array} \qquad (5.3)$$

This version might be read as "to establish that a new input is consistent with a current set of constraints, show that the input is derivable from the current database and the relevant constraints." Of course this reading is a simplification, since the axiom says that this should be done by attempting to find an inconsistency, and then restore consistency by a procedure *AnalyseFailureRestoreConsistency*. One difference between formulas (5.1) and (5.3) is that the former shows that a new assertion is consistent by re-deriving only the relevant constraints. In Kowalski's version, isolation of constraints is done implicitly by assuming that variable names are interpreted as types, and that *Demo* and *AnalyseFailRestoreConsistency* achieve their intended effect by using auxiliary procedures to collect constraints and make database changes. The DLOG specification provides a way of determining which constraints are relevant. Another advantage of (5.1) over (5.3) is implementation clarity: (5.3) is closer to implementation using a *procedural* interpretation of Horn clauses, but (5.1) more clearly shows the relationships between assertions, constraints, and databases.

Much of the effort in implementing DLOG is in specifying Prolog meta predicates for achieving

---

[22] [Kowalski81, p. 20].

the effect intended in (5.1). For example, if we assume an appropriate axiomatization of two new predicates $all(*,*,*)$ and $update(x)$, then (5.2) might be rendered as:

$$\begin{aligned} assertable(x) \subset\ & assertion(x) \\ & \wedge all(y, constraint(y) \wedge relevant(x,y), X) \\ & \wedge derivable(X) \\ & \wedge update(x) \end{aligned} \tag{5.4}$$

First, we assume that there is always one current theory that is implicitly identified, so all references to it (i.e., in *assertable*, and *derivable*) are dropped. Next, we use a meta predicate *all* (cf. [Clocksin81, Pereira83]) to isolate the relevant constraints and bind them to the set variable X. We expect that *all* will use its second argument as a generator, to construct the set of all relevant constraints X. The final assumption is that *derivable* will be axiomatized so that when given a set X of formulas, it will attempt to derive each formula of that set. The final modification is the addition of a database predicate *update* that will add the "assertable" formula x to the implicitly specified database. Under these assumptions, (5.4) closely mimics Prolog syntax, except for the set variable X.

5.1.2. Standard extensions to Prolog

The Prolog implementation of DLOG uses some meta predicates that have, by now, become established as "standard" extensions (e.g., see [Clocksin81]). These include a specification of the meta predicate *all* used to mimic universal quantification in goal statements, and special axioms to mimic finite disjunction. DLOG uses the following:

$$all(lambda(*x, *expression), *extension)$$

The *all* meta predicate represents a relation on lambda terms and sets of individuals. It is similar to CProlog's $setof(x,P,S)$ meta predicate that is used to create a set S of all individuals x that satisfy the Cprolog goal P [Pereira83]. This meta predicate is derivable when the extension of the complex predicate described by the lambda expression is bound to the variable **extension*. In the implementation, suitable modifications enforce the sort constraint that treats the extension as a DLOG set term.

The DLOG disjunctive connectives are described in Prolog as follows:

$*x \mid *y$ <- $*x$
$*x \mid *y$ <- $*y$

and

$*x$! $*y$ <- $*x$ & $\neg *y$
$*x$! $*y$ <- $\neg *x$ & $*y$

The implementation substitutes '|' and '!' for 'v' and '⊻', respectively. The Prolog implementation treats the operators as meta predicates: inclusive alternation requires the derivation of at least one alternative, while the exclusive case requires proof of one disjunct and the negation of the other. As discussed below (§§5.2, 5.2.3), this implementation of disjunctive operators does not provide a general facility for interpretating disjunctions embedded within descriptions. Note also that the definition of the alternatives for the disjunctions rely on finite failure of both subgoals of the first alternative.

5.2. A DLOG implementation

The terminal session of fig. 2.5 was produced with the most recent DLOG implementation written in Waterloo IBM Prolog [Roberts77]. The source code for this implementation is included as Appendix 1.

The current prototype has only weak versions of negation and disjunction, but it is capable of interpreting each form of DLOG descriptive term. At least one instance of each term occurs in the DDB, as the motivation for their construction arose in that domain. A complete listing of the DDB is give in Appendix 4.

Several features have been provided in response to traditional DBM system facilities, but others are unique to DLOG. Examples of the former include a data dictionary facility, a facility for interpreting application commands (i.e., "data manipulation language" interface), and a transaction processor. In addition, the system augments the traditional data dictionary with a topic structure [Goebel77, Goebel78, Schubert79]. Topic predicates are included to organize predicates of the data dictionary into categories. A user can use the topic structure to browse the database contents by topic, to get a "feeling" for the kinds of information stored.

A feature unique to this implementation is a facility for heuristic interpretation of various kinds of queries. The system is capable of invoking heuristics when the logical interpretation of a DLOG query fails. This is DLOG's concession to "procedural attachment" (cf. FRL, KRL), but the DLOG framework is such that heuristic processing is initiated only after deductive processing fails. Further details of this proof heuristic based on partial matching are given in §5.4.

5.2.1. Implementation syntax

The restricted syntax of complex terms given in §3.3.1 are only part of the way to our desired specification in a standard single-sorted first order logic. The final step to an implementation is the translation of the 3-sorted syntax to the single-sorted syntax of Prolog. In general, the translation of an n-sorted language to a single-sorted language is done by rewriting sentences of the form

$$\forall x_\tau . \Phi(x_\tau)$$
$$\exists x_\tau . \Phi(x_\tau)$$

as

$$\forall x . \tau(x) \supset \Phi(x)$$
$$\exists x . \tau(x) \wedge \Phi(x)$$

respectively, where x_τ is a variable of sort τ. Intuitively, the sort τ is translated as a unary predicate of the theory described in the single-sorted language.

For example, in the case of sets we can use a unary predicate *set* to distinguish terms of sort 'set', and expand

$$\exists X_1 . \Phi(a, X_1)$$

as

$$\exists x_1 . set(x_1) \wedge \Phi(a, x_1)$$

The latter is rather more complicated than the former, but gives the intended meaning in a standard single-sorted way.

As a further example, the translation of the sentence

$$\forall X_1 \forall x_2.[[x_2 \in X_1 \supset \Pi(x_2)] \wedge \Psi(X_1)] \supset \Phi(X_1)]$$

to a single-sorted language is

$$\forall x_1.[set(x_1) \supset [\forall x_2.[x_2 \in x_1 \supset \Psi(x_2)] \wedge \Pi(x_1)] \supset \Phi(x_1)]$$

This simply removes the set variable X_1. Similar translations are made for the sort *lambda*.

This implementation syntax is summarized in the table below. Many of the syntactic constructs are concessions to standard keyboard devices. The implementation syntax will be used in the rest of this chapter.

lexical item	specification	implementation
individual variables	$x_1, x_2, \ldots$	*x1, *x2, ...
set variables	$X_1, X_2, \ldots$	set(*x1), set(*x2), ...
set constants	$\{\alpha \wedge \beta \wedge \ldots \wedge \delta\}$	set(α & β & ... & δ)
definite individuals	$\iota x.\Phi(x)$	the(*x,Φ(*x))
indefinite individuals	$\epsilon x.\Phi(x)'$	an(*x,Φ(*x))
definite sets	$\{x : \Phi(x)\}$	set(*x:Φ(*x))
indefinite sets	$\{x, X : \Phi(x) \wedge \Pi(X)\}$	set(*x1;set(*x2):Φ(*x1)&Ψ(set(*x2)))
	$\{\alpha R \beta R \ldots R \delta\}$[†]	set(α R β R...R δ)[‡]
lambda expressions	$\lambda x.\Phi(x)$	lambda(*x,Φ(*x))
implications	$C \subset A_1 \wedge A_2 \wedge \ldots \wedge A_n$	C <- A_1 & A_2 & $\cdots$ & A_n
constraints	$A \supset C_1 \wedge C_2 \wedge \ldots \wedge C_n$	A -> C_1 & C_2 & $\cdots$ & C_n
queries	$?C_1 \wedge C_2 \wedge \ldots \wedge C_n$	<-C_1 & C_2 & $\cdots$ & C_n

† R is any of $\wedge$, $\vee$, or $\dot{\vee}$, and one of $\vee$ or $\dot{\vee}$ must appear.

‡ R is any of &, |, or !, and one of | or ! must appear.

Table 5.1 Specification and Implementation Syntax

5.2.2. System structure

From a system implementor's viewpoint, the DLOG system is a collection of independent programs, collected within a framework that routes information among these programs and controls their operation. We will refer to these programs as DLOG processors, and describe the system in terms of their relationships. The major processors and their basic relationships are given in fig. 5.1.

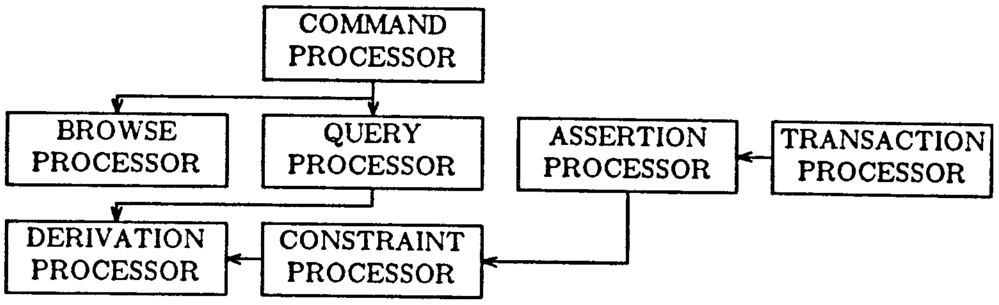


Figure 5.1 DLOG processor architecture

Directed edges show possible flow of control, e.g., the transaction processor may require the services of both the assertion processor and constraint processor. Each processor can be viewed as a Prolog axiomatization of the appropriate relation.

User inputs are filtered by the command processor and, depending on their content, are forwarded to the appropriate sub-processors. Each processor may set various global flags to select optional processing in dependent processors. This is most useful when combined with Prolog's failure-driven backtracking: each "call" is a goal request, and failing goals return control to their initiating processor to pursue an appropriate alternative. However, it is not exemplary logic programming style — it is done here for efficiency and expedience.

The DLOG implementation could be viewed as a single complex Prolog theory, however a loaded application database is distinguished from implementation axioms by using the DLOG data dictionary. Furthermore, since Kowalski requires that meta and object databases be clearly distinguished,[23]

[23] [Kowalski81, p. 21].

application predicates are enumerated within the application's data dictionary. The implementation can use this dictionary to distinguish object theory axioms from meta theory axioms. Further data dictionary details are provided in §5.5.4.

5.2.3. DLOG derivability

The foundation of the DLOG implementation is the Prolog axiomatization of the *derivable* relation. This Prolog implementation of *derivable* performs DLOG query evaluation, and uses Prolog's meta variable facility (§5.1) only for more efficient derivation of system implementation predicates. As discussed in §3.4, the Prolog implementation of DLOG's derivable predicate represents the relation $\vdash$ on DLOG databases and queries. The implementation of *derivable*(DB,Q) given here is slightly more elaborate in that it uses extra information about the form of a query to determine if the general specification of derivable (§3.4) is required. In fact, the implementation actually includes three distinct proof strategies, the selection of which is controlled by data dictionary information. The meta predicate *i−derivable* corresponds most closely to the specification of *derivable* given in §3.4, except that the connectives '&', '!', and '|' are interpreted elsewhere. Note also, that since there is at most only one application database at any one time, the *DB* argument in *derivable*(DB,Q) is left out. The three strategies are:

1) derivation of literals without set predicates (s-derivable). As mentioned §4.3.4, there can be some ambiguity in the interpretation of assertions involving sets as terms. For example, if '$set(a\&b)$' is properly treated as a set constant, the formula '$P(a)$' does not follow from the formula $P(set(a\&b))$. This implementation of DLOG will use s-derivable to attempt such a derivation. The ambiguity about whether the assertion $P(set(a\&b))$ actually means $P(a)\&P(b)$ is removed by a *set−predicate* assertion in the DLOG data dictionary (§5.3.4). This is the way in which a user can specify a "weak typing" of predicates, as suggested in §4.3.4. The s-derivable strategy is selected as follows:

```
derivable(*atomic-goal) <-
        get-predicate(*pname,*atomic-goal)
      & ¬set-predicate(*pname)
      & s-derivable(*atomic-goal).
```

A good example of this strategy's use is the frequent derivation of the *satisfies* predicate. Recall (§4.1.2) that *satisfies*(α,β) can be derived when α is a DLOG constant and β is a DLOG lambda expression applicable to α and derivable from the current database. The *satisfies* predicate is axiomatized as

```
satisfies(*x,lambda(*x,*goal)) <-
        derivable(*goal)
```

but is frequently used in queries with a set of lambda expressions. For example,

*satisfies(John,set(*x:course-req(BScMajorsCS,first,*x)))*

asks if "John has satisfied the set of all first year course requirements for BScMajorsCS." When a goal involves a non-*set-predicate* but includes a set term, the s-derivable derivation strategy expands the set description and applies the predicate to each element of the set's extension. In other words, the s-derivable strategy uses data dictionary information to decide how to interpret the manipulation of predicates that have set arguments. Notice that this strategy doesn't make sense for goals with multiple set terms, since the application of a predicate to a Cartesian product of multiple set extensions is nonsense. Notice also, that this strategy is *not* part of the DLOG data model specification — it is an implementation strategy that helps streamline the user interface.

2) derivation of correctly typed arguments (i-derivable). The i-derivable proof strategy is used in most cases, where either the goal includes a set term (and has been typed as such), or doesn't contain a set term but cannot be derived directly. In other words, when predicates are used as specified by the typing of the data dictionary, this proof strategy is selected. This strategy is the most general, and is the one from which extended unification is used to match DLOG complex terms. It is selected as follows:

```
derivable(*atomic-goal) <-
        get-predicate(*pname,*atomic-goal)
      & set-predicate(*pname)| ¬contains-set-term(*atomic-goal)
      & i-derivable(*atomic-goal).
```

3) direct derivation (d-derivable). Any literal of a DLOG query that uses a DLOG system predicate (e.g., *satisfies*, *extension*) or a Prolog system predicate (e.g., *SUM*, *AX*) can be derived directly by Prolog rather than be simulated with the more general strategies 1) and 2) above. This strategy is selected when the predicate of the current goal has the *system-predicate* or *prolog-predicate* properties:

```
derivable(*atomic-goal) <-
        get-predicate(*pname,*atomic-goal)
      & system-predicate(*pname) | prolog-predicate(*pname)
      & *atomic-goal.
```

5.2.3.1. DLOG unification

Though the referential interpretation of descriptions (§4.2) requires manipulation of DLOG's descriptive terms at assertion time, much of the required special treatment can be done at query evaluation time at the term matching level. Instead of extending the *derivable* axiomatization to handle the standard rewritings of descriptions (§3.2.1), the unification algorithm can be augmented to provide the correct matching of descriptive terms. In a sense, some of the complexity of derivation is off-loaded to the pattern-matcher (cf. [Reiter75]).

The idea of extending a resolution proof procedure's power by augmenting unification was first suggested by Morris [Morris69], who proposes that equality be dealt with by so-called "E-unification." Other proposals include Stickel [Stickel75], Morgan [Morgan75], Kahn [Kahn81] and Kornfeld [Kornfeld83]. Kahn proposes unification as the *only* computation mechanism in a complete programming language. Kornfeld's proposal is more coherent; he intercepts normal unification to apply a simple theory of equality that permits proofs like

$$6=succ(5),\ P(6) \vdash P(succ(5))$$

More interesting (but less coherent) is the representation language KRL [Bobrow77b, Bobrow77a, Bobrow79], which relies on a comprehensive and complex "mapping" process on several different kinds of object descriptions called "descriptors." To illustrate how DLOG might provide insight into the mapping operation of the KRL language, a version of Hayes' [Hayes80] logical rewriting of KRL examples appears in Appendix 3.

Returning to the handling of descriptive terms by augmenting unification, we cite Rosenschein on the advantage of embedded terms:

> ...the data object is kept small and "hierarchical" so that where an exhaustive match must be performed, failure can occur quickly. That is, deep, heterogeneous structures are preferred to broad, homogeneous structures. For example, {(){()()}} is better than {{}{}{}{}}.[24]

We view Rosenschein's claim as support for the interpretation of descriptions as embedded terms, rather than as their contextual definition by rewriting.

The DLOG unification algorithm is invoked by the DLOG *derivable* predicate, similar to the way Prolog's derivation procedure uses a built-in unification algorithm. Intuitively, whenever a unification must be performed and there are special DLOG terms to be matched, standard unification is intercepted, and DLOG unification is used. For example, suppose that the two terms

$$an({}^{*}x, \Phi({}^{*}x))$$

and

$$Fred$$

are to be unified. The applicable DLOG *unify* axiom is

$$\begin{aligned}&unify(an({}^{*}x,\Phi({}^{*}x)),{}^{*}y) <\text{-}\\ &\qquad individual-constant({}^{*}y)\\ &\qquad \&\ apply(lambda({}^{*}x,\Phi({}^{*}x)),{}^{*}y)\end{aligned}$$

where *apply* is an internal variation of the *satisfies* predicate (§3.5). Here the axiom says that the two terms unify if the property Φ of the indefinite description can be verified for the constant term

---

[24] [Rosenschein78, p. 534].

Fred. This is the implementation of equality axiom schema (3.24) of §3.4.2.

Notice here that the relationship to a functional model of computation is strong because the equals "test" on the indefinite term and individual is expressed as the application of a lambda constant. As mentioned in §3.4.1, the axioms for the predicate "=" could be viewed as the specification for a kind of functional model of computation. As Goguen and Meseguer [Goguen84] show, the derivation of an equality theorem can be treated as the computation of a solution for a set of equalities. For example, Goguen and Meseguer's Eqlog language includes a theory of equality that subsumes Prolog's theory of equality [Emden84]. In Eqlog, equality proofs are provided by using an equation solving procedure called "narrowing." In general, narrowing is not decidable so there is much debate over the claimed advantages of Eqlog's approach to equality. For example, Kowalski[25] suggests that, because first order unification does have an algorithm, equality extensions should be expressed as axioms about the predicate "=," instead of by a replacement like narrowing.

The DLOG unification definition uses an organization similar to the LOGLISP system of Robinson and Sibert [Robinson82a, Robinson82b]. LOGLISP consists of a logical proof theory embedded within LISP, and allows the invocation of LISP by the theorem-prover, and the theorem-prover by LISP. Similarly, the DLOG *derivable* procedure can invoke the standard Prolog proof procedure (via the i-derivable proof strategy, §5.2.3), and both are recursively accessible from within DLOG's unification matcher.

In general, the correct "unification" of the DLOG extensions requires a derivation procedure more powerful than that provided by Prolog. For example, the equivalence of two lambda expressions, e.g.,

$$lambda(*x,\Phi(*x))$$
$$lambda(*x,\Psi(*x))$$

can only be made if it can be shown that

[25] Invited address "Problems of logic programming," to the Second International Conference on Logic Programming, Tuesday July 3, 1984, Uppsala University, Sweden.

$$\Phi(^*x) \equiv \Psi(^*x)$$

follows from the current database. The current DLOG unification procedure uses a meta predicate *eqlambda* to derive this equivalence. By using a local context mechanism, the equivalence can be established by proving

$$\Phi(^*x) <\!\!- \Psi(^*x)$$
$$\Psi(^*x) <\!\!- \Phi(^*x)$$

This is a case where disjunctive terms would require a more general proof mechanism, since a proof of

$$\Phi(^*x) \vee \Psi(^*y) \equiv \Phi(^*x) \vee \Psi(^*y)$$

cannot be handled by the current implementation.

The local context mechanism provides the necessary temporary hypothesis used in the standard strategy for deriving implications: to derive $\Phi(^*x) \rightarrow \Psi(^*x)$, we first generate a new constant symbol, say α, then assume $\Phi(\alpha)$ and attempt to derive $\Psi(\alpha)$. Kowalski [Kowalski81] proposes a similar mechanism for pursuing Prolog derivations using "auxiliary" axioms. The DLOG implementation includes general facilities for using temporary contexts, and for creating and destroying contexts by name. These facilities are used in many other processors, e.g., the transaction processor (§5.4.4.4).

Now follows the top-level specification for each possible DLOG unification task. These specifications are intended to the implementation of the equality axioms for complex terms (§3.4.2).

1. $unify(c_1,c_2)$

   As in standard unification, two individual constants c_1, c_2 unify if and only if they are identical symbols. In Prolog this is determined by comparing the string representation of constants. In Prolog we have

   $unify(c_1,c_2)$ <-
   $\quad EQ(c_1,c_2)$.

2. $unify(^*x_1,c_1)$

   Again as in standard unification, and individual variable $^*x_1$ and an individual constant c_1 unify with the result that c_1 is substituted for $^*x_1$. In Prolog we write

```
unify(*x_1,c_1) <-
      bind(*x_1,c_1).
```

where *bind* is implemented as the atomic assertion *bind*(*x,*x).

3. *unify*(*the*(*x,Φ(*x)),c_1)

This combination of definite individual and individual constant can arise only when the definite individual comes from a query. This is because definite individuals in assertions are processed at assertion time (see §4.3.1). The two terms here are DLOG unifiable if and only if the extension of the predicate Φ in the current database is a singleton set with c_1 as its sole member. The implementation is

```
unify(the(*x,Φ(*x)),c_1) <-
      extension(set(*x:Φ(*x)),set(c_1)).
```

where *extension* is the DLOG system predicate that computes the extension of any DLOG set term. The above goal requests that *extension* verify that the extension of Φ in the current database is the DLOG set with one member c_1.

4. *unify*(*the*(*x,Φ(*x)),*an*(*x,Ψ(*x)))

The successful unification of an indefinite individual and a definite individual is similar to case 3, except that, in addition to demonstrating the uniqueness of a referent for the definite individual, the sentences Φ(*x) and Ψ(*x) must also be shown equivalent. In this case, the *eqlambda* predicate described above is used:

```
unify(the(*x,Φ(*x)),an(*x,Ψ(*x))) <-
      eqlambda(lambda(*x,Φ(*x)),lambda(*x,Ψ(*x)))
    & extension(set(*x:Φ(*x)),set(*y))
    & individual-constant(*y).
```

The *extension* predicate is used to find the set of all individuals that satisfy the body of the definite individual, and then *individual*−*constant* ensures that the extension found is a set consisting of a single individual.

5. *unify*(*an*(*x,Φ(*x)),*an*(*x,Ψ(*x))

The unification of two indefinite individuals simply requires the use of the *eqlambda* predicate:

unify$(an(^*x,\Phi(^*x)),an(^*x,\Psi(^*x))$ <-
eqlambda$(lambda(^*x,\Phi(^*x)),lambda(^*x,\Psi(^*x)))$.

6. *unify*$(^*x,an(^*x,\Phi(^*x))$

Since an indefinite term is treated like a Skolem constant (see §4.3.2), this case is the same as case 2:

unify$(^*x,an(^*x,\Phi(^*x)))$ <-
bind$(^*x,an(^*x,\Phi(^*x)))$.

7. *unify*$(lambda(^*x,\Phi(^*x)),lambda(^*x,\Psi(^*x)))$

Lambda terms are unifiable if and only if their defining sentences are equivalent:

unify$(lambda(^*x,\Phi(^*x)),lambda(^*x,\Psi(^*x))$ <-
eqlambda$(lambda(^*x,\Phi(^*x)),lambda(^*x,\Psi(^*x)))$.

8. *unify*$(set(^*x_1),set(^*x_2))$

As for standard unification, set variables unify with any other set term by simple substitution. These substitutions include cases 8, 9, 10, and 11.

unify$(set(^*x_1),set(^*x_2))$ <-
bind$(^*x_1,^*x_2)$.

9. *unify*$(set(^*x),set(c_1\&c_2\& \cdots \&c_n))$

The term $set(c_1\&c_2\& \cdots \&c_n)$ is a set constant (see §3.1.1):

unify$(set(^*x),set(c_1\&c_2\& \cdots \&c_n))$
bind$(^*x,c_1\&c_2\& \cdots \&c_n)$

10. *unify*$(set(^*x),set(^*y{:}\Phi(^*y)))$

Definite sets are simply bound to the set variable, like set constants. In general, the expansion (e.g., by computing an extension) of set descriptions is avoided as long as possible.

unify$(set(^*x),set(^*y{:}^*x))$ <-
bind$(^*x,^*y{:}\Phi(^*y))$.

11. *unify*$(set(^*x),set(^*y;set(^*z){:}^*x)\&\Psi(set(^*z))))$

Recall from §3.1.1 that both an individual and set variable are required in the indefinite set term.

$unify(set(^*x), set(^*y; set(^*z): {}^*x) \& \Psi(set(^*z)))$ <-
  $bind(^*x, {}^*y; set(^*z): {}^*x) \& \Psi(set(^*z)))$.

The remaining cases include those which require by far the most computation to unify. This is because demonstrating the equivalence of set descriptions often requires testing equivalence of their extensions.

12. $unify(set(c_1 \& c_2 \& \cdots \& c_n), set(^*x : \Phi(^*x)))$

A set constant and definite set are unifiable whenever their extensions are equal.

$unify(set(c_1 \& c_2 \& \cdots \& c_n), set(^*x : \Phi(^*x)))$ <-
  $extension(set(^*x : \Phi(^*x)), {}^*ext)$
  $\& \; eqext(set(c_1 \& c_2 \& \cdots \& c_n), {}^*ext)$.

13. $unify(set(c_1 \& c_2 \& \cdots \& c_n), set(^*x ; set(^*y): \Phi(^*x)) \& \Psi(set\, ^*y)))$

This situation is similar to case 6 for individuals; we must show that $\Phi(^*x)$ is derivable for each individual constant c_i, and that $\Psi(set(^*y))$ is derivable for $set(^*y) = set(c_1 \& c_2 \& \cdots \& c_n)$. Here another DLOG built-in predicate *map* provides an iterative version of the previously used *apply* predicate:

$unify(set(c_1 \& c_2 \& \cdots \& c_n), set(^*x ; set(^*y): \Phi(^*x)) \& \Psi(set\, ^*y)))$ <-
  $map(lambda(^*x, \Phi(^*x)), set(c_1 \& c_2 \& \cdots \& c_n))$
  $\& \; apply(lambda(set(^*y), \Psi(set(^*y))), set(c_1 \& c_2 \& \cdots \& c_n))$.

14. $unify(set(c_1 \& c_2 \& \cdots \& c_n), set(c_{n+1} \& c_{n+2} \& \cdots \& c_{n+m}))$

Two set constants are equivalent if they are extensionally equivalent. In this case, the *eqext* predicate will sort the sets into a normal form (ASCII collating sequence) to determine their equivalence:

$unify(set(c_1 \& c_2 \& \cdots \& c_n), set(c_{n+1} \& c_{n+2} \& \cdots \& c_{n+m}))$ <-
  $eqext(set(c_1 \& c_2 \& \cdots \& c_n), set(c_{n+1} \& c_{n+2} \& \cdots \& c_{n+m}))$.

15. $unify(set(^*x : \Phi(^*x)), set(^*y : \Psi(^*y)))$

The equivalence of descriptions is tested by using *eqlambda*:

$unify(set(^*x:\Phi(^*x)),set(^*y:\Psi(^*y)))$ <-
  $eqlambda(lambda(^*x,\Phi(^*x)),lambda(^*y,\Psi(^*y)))$.

16. $unify(set(^*x;set(^*y):\Phi(^*x)\&\Psi(set(^*y))),set(^*z;set(^*w):\Omega(^*z)\&\Gamma(set(^*w))))$

The equivalence of indefinite sets is tested by using *eqlambda*:

$unify(set(^*x;set(^*y):\Phi(^*x)\&\Psi(set(^*y))),set(^*z;set(^*w):\Omega(^*z)\&\Gamma(set(^*w))))$ <-
  $eqlambda(lambda(^*x,\Phi(^*x)),lambda(^*z,\Omega(^*z)))$
  & $eqlambda(lambda(set(^*y),\Psi(set(^*z))),lambda(set(^*w),\Omega(set(^*w))))$.

17. $unify(set(^*x:\Phi(^*x)),set(^*y;set(^*z):\Psi(^*y)\&\Omega(set(^*z))))$

The equivalence test here is slightly more complex, as unification can succeed only if the extension of the indefinite set has but one extension; i.e., a set each of whose members satisfies the property specified in the definite set:

$unify(set(^*x:\Phi(^*x)),set(^*y;set(^*z):\Psi(^*y)\&\Omega(set(^*z))))$ <-
  $eqlambda(lambda(^*x,\Phi(^*x)),lambda(^*y,\Psi(^*y)))$
  & $extension(set(^*x,\Phi(^*x)),^*ext)$
  & $apply(lambda(set(^*z),\Omega(set(^*z))),^*ext_1)$
  & $all(lambda(^*x,extension(set(^*y;set(^*z):\Psi(^*y)\&\Omega(set(^*z))),^*x)),^*ext_2)$
  & $cardinality(^*ext_2,1)$.

The properties required for equivalence are tested in such a way as to avoid computing the extension of the indefinite set if possible. As these set objects are the most complex in DLOG, it is expected that this kind of match will rarely succeed; therefore the properties that are easier to compute are tested first. The final computation verifies that the extension of the indefinite set is unique.

5.2.4. DLOG processors

Having described the heart of the DLOG implementation, the DLOG derivation processor, we turn to the six other major processors that appeared in fig. 5.1.

5.2.4.1. Command processor

DLOG's command processor is the system's user interface. It consists of axiomatizations of three main predicates:

> (1) *input(*string)*,
> (2) *parse(*input,*output)*, and
> (3) *assertion(*ax)*.

The predicate *input* expects either a command, a query, or an assertion. Using the meta variable facility, *input* uses command names to invoke the appropriate processor, e.g.,

```
input(*command) <- *command.
```

will invoke the predicate *transaction* if the user types the command "transaction." Queries are written in the standard Prolog/MTS fashion with a '<-' prefix and are simply forwarded to the query processor:

```
input(<-*x) <- query(*x)
```

When the input is not a command or a query, it is assumed to be an assertion. The *parse* and *assertion* predicates are invoked to apply input macros, to verify syntax, and to perform any required assertion time inference.

A feature of the command processor is its use of the syntax checking predicate *assertion*. The *assertion* predicate can be viewed as the start symbol of a grammar that describes the DLOG assertion syntax. For example we might have

```
assertion := atom
           | implication
           | constraint
```

as a production of such a grammar; in Prolog we can write this production as

```
assertion(*x) <- atom(*x)
               | implication(*x)
               | constraint(*x)
```

with the bonus that, when we invoke *assertion* with a collection of terms, the Prolog derivation procedure does a top-down parse of the input. Note that the assertion predicate has only one argument, and is distinct from the DLOG database maintenance predicate *assertable* (§§3.6, 5.2).

During the derivation of *assertion* goals (i.e., parsing), the input is broken into lexical items in the normal way. The *parse* predicate can exploit this processing in the following way: when transformations must be applied to particular components of an assertion, the axioms for the predicate of the corresponding non-terminals are augmented with an extra conjunctive goal that makes the appropriate transformation. The *parse* predicate is a binary relation on inputs and transformed inputs; it sets a flag that triggers the appropriate transformations when relevant syntactic items are encountered. This is a matter of computational efficiency distinct from any logical consideration. The saving can be significant, since each item is decomposed and parsed once, regardless of the number of transformations.

As an example, consider the referential interpretation of definite individuals (§4.3.1). Ordinarily, the interpretation of an assertion like $\Phi(the(^*x,\Psi(^*x)))$ might begin after the syntax had been verified. Instead, the command processor uses the *parse* predicate to set a flag that triggers the *check−input* predicate added to the grammar "production" for definite individuals:

```
definite−individual(the(*x,*e)) <-
                    individual−variable(*x)
                   & clause(*e)
                   & check−freevar(*x,*e)
                   & check−input(the(*x,*e))
```

The first three conjuncts of the antecedent verify the DLOG syntax of definite individual descriptions, and the last one will perform the referential check if requested by *parse*. The logical gist is that, when a transformation is required, the input or parser axioms augment the current implementation theory with an assertion about the current input. The *check−input* predicate is always true, but in the augmented theory will apply the required transformations and update the current input assertion with the transformed input. If the parse is successful, the transformed input can be retrieved. When no current inputs are asserted the parse proceeds normally, so any other processor can use the *assertion* predicate to verify syntax without fear of transformations being applied.

5.2.4.2. Assertion and constraint processors

As described in §3.3.1, DLOG constraints have the form A->B where A is an atomic assertion and B is a body. The actual implementation of integrity maintenance is just as §3.6 suggests: each constraint of the form above is treated as an assertion about *assertability*, viz.

assertable(A) <- *derivable*(B)

The current set of integrity constraints are taken as a Prolog axiomatization of the predicate *assertable* on DLOG atomic assertions. To enforce integrity, the DLOG implementation treats each new assertion as a query — as a request to establish the "assertability" of the assertion.

The implementation does not require that each assertion be made by explicitly requesting that assertability be demonstrated. Instead, any user input that is not a DLOG system command is passed on to the assertion processor to be tested for assertability. The top level program for the assertion processor is:

```
assertable(*ax) <-                                                  (5.5)
        all(lambda(*con,AX(*ant->*con)
                         & unify(*ax,*ant)),*constraints))
     & derivable(*constraints)
     & ADDAX(*ax).
```

Compare this with formula (5.4) above. Recall that the *all* predicate will compute the extension of its first argument, a unary lambda expression, and bind the result to its second argument (§5.1.2). In this case, the lambda expression describes the set of all terms of the form **ant->*con* whose antecedent part **ant* unifies with the new assertion **ax*. The *AX* predicate is a Prolog predicate (cf. *clause* in DEC 10 or Edinburgh CProlog) that is used to retrieve axioms from the current Prolog database. In this case *AX* is being used to retrieve all integrity constraints.

As mentioned in §3.6, we would rather not rederive all the constraints for each new assertion but only those "relevant" to the new assertion. In this implementation the relevance of a constraint is determined by the *unify(*ax,*ant)* portion of the lambda expression in sentence (5.1). In other words, the set of all constraints identified by *AX(*ant->*con)* is further restricted by requiring that the antecedent **ant* of each constraint unifies with the new assertion **ax*.

A simple example will illustrate. Consider the following constraints:

1) *course-enrolled(*s, *c) ->*
*course(*c)*
*& (registered(*s, *p, *) | eligible-for-admission(*s, *p))*
*& satisfied(*s, set(*x:course-prerequisite(*c, *x)))*
*& student-program-contribution(*s, *p, *c).*

*2) course-enrolled(*s, CS786) ->*
*has-special-permission(*s, CS786).*

*3) recommended(*p, *y, *c) ->*
*program-contribution(*p, *y, *c).*

Constraint 1) applies to all new assertions that enroll a student **s* into some course **c*. The second constraint 2) applies to any student who attempts to enroll in *CS*786—it is a special consideration particular to that course. The last constraint, 3), deals with assertions about recommended courses; it requires that any course being recommended for some year **y* in program **p* must be a contribution to that program.

Now consider the set of constraints retrieved by the *all* predicate of sentence (5.5) when the assertion *course−enrolled*(*Kari*,*CS*115) is bound to the variable **ax*. The *AX*(**ant*->**con*) portion of the lambda expression will match and retrieve all of the above constraints 1), 2), and 3), but the *unify*(**ax*,**ant*) portion will filter constraints 2) and 3). Constraint 3) is obviously not relevant: the predicates are different. Constraint 2) is rejected, not because the predicates conflict, but because the course names conflict. Constraint 3) is relevant only to assertions about enrolling in the course *CS*786. Now the assertability of *course−enrolled*(*Kari*,*CS*115) will be determined by the derivability of

$$\begin{aligned} & course(CS115) \\ & \& \ (registered(Kari,{}^*p,{}^*) \mid eligible-for-admission(Kari,{}^*p)) \\ & \& \ satisfied(Kari,set({}^*x{:}course-prerequisite(CS115,{}^*x))) \\ & \& \ student-program-contribution(Kari,{}^*p,CS115). \end{aligned} \tag{5.6}$$

Notice that the assertion *course−enrolled*(*Kari*,*CS*786) would simply add the extra condition *has−permission*(*Kari*,*CS*786) to the appropriate version of (5.6).

5.2.4.3. Query processor

The query processor is quite simple since DLOG's *derivable* predicate does most of the work. As with *assert*, the *parse* predicate of the command processor verifies query syntax. The query processor must only forward a query to the *derivable* predicate:

```
query(<-*q) <- derivable(*q)
```

In addition *query* will take arbitrary set descriptions and compute their extensions:

```
query(<-set(*x)) <-
    extension(set(*x),*y)
  & write(*y).
```

For example,

```
<-set(*x:program(BScMajorsCS,first,*x))
```

will return the set of program requirements for a Bachelor of Science in Computer Science for the first year. The *extension* predicate is the same one used in §5.2.3.1 during DLOG unification.

5.2.4.4. Transaction processor

Since constraints are verified for each assertion, situations can arise where one ordering of assertions results in a set of consistent updates, but an alternative ordering results in rejection of the same assertions. For example, assuming an empty DLOG database, the assertions

```
course(CS115)
course-no(CS115,115)
course(*x) -> course-no(*x,*y) & integer(*y)
```

will be accepted, but the same assertions in the order

```
course(CS115)
course(*x) -> course-no(*x,*y) & integer(*y)
course-no(CS115,115)
```

will not. In fact, without some mechanism for "packaging" a collection of updates and checking their consistency simultaneous, the integrity constraint in the latter list of three new assertions will be rejected, as it is cannot be satisfied in the database consisting only of the assertion '*course*(*CS*115)'.

This is because there is no way to verify that all courses have a course number before the course number for *CS*115 is asserted. This integrity checking on individual updates is especially annoying when loading databases — if a data base is globally consistent, then assertion order should not affect integrity enforcement.

In response to this problem, DLOG has a transaction processor that will accept sets of assertions before applying integrity constraints. When transaction mode is terminated, all relevant constraints are checked, and *all* assertions are revoked if any integrity constraint is violated.

The transaction mechanism works by augmenting the DLOG data base with each new assertion while maintaining a list of those assertions in the implementation database (i.e., an internal database separate from the application database). When the transaction ends, normal constraint processing is done for each assertion recorded in the internal database. If a constraint fails, the appropriate assertions are removed from the DLOG database. Otherwise, the internal database list is deleted and normal processing continues.

5.2.4.5. Browser

When the contents of an application database are unfamiliar, it is useful to inspect the database directly, rather than with a barrage of potentially meaningless or irrelevant queries. The browse processor is a system feature, independent of the DLOG data model specification, that allows a user to browse the application database much in the same way a relational database user might list tables of relations. The browser performs no (object level) inference; it merely retrieves axioms as directed by the user. When invoked with the *browse* command, the user is presented with the following menu:

```
1: Topics
2: Constraints
3: User predicates
4: System predicates
5: Enter predicate
6: Enter skeleton

Selection?
```

After selecting an alternative, the user is guided through the relevant information a bit at a time, until he has seen enough, or until no further entries of that category exist.

The first and third menu alternatives (topics and user predicates) let a user peruse the application database's data dictionary. Its format is described below (§5.5.4).

The topic browser takes arbitrary names or strings as input, and attempts to produce a list of user predicates relevant to that topic. Relevant topics are determined by a synonym dictionary and a topic hierarchy [Goebel77, Goebel78, Schubert79], both included as a portion of the application data dictionary (see §5.3.4 for an example).

The "user predicate" selection steps through the application database's list of predicates. Similarly, option 4 steps through the user accessible DLOG system predicates recorded in an internal system data dictionary.

The fifth option prompts the user for a predicate name. If that name can be identified, its axioms are presented. The remaining options (2 and 6) will prompt for atomic clauses, and browse axioms that match the user-specified clause. For example, after the menu appears and option 2 is selected, the input

$$course-req(BScMajorsCS,*,*)$$

will browse all relevant constraints for the *course−req* predicate whose first argument is *BScMajorsCS*.

5.3. Traditional Database Management facilities

The DLOG system is an experimental program, too uncivilized and brittle for real world use. However, its usefulness as an experimental tool is greatly enhanced by several facilities motivated by traditional Database Management (DBM) implementations. These include a data sublanguage embedded within a general purpose programming system, a transaction facility, an integrity constraint facility, and data dictionary support.

Except for integrity constraints, these facilities are implementation adjuncts to the DLOG data model. The data model specification has nothing to say about the correctness or semantics of any implementation-dependent feature. Nevertheless, these facilities are an important part of any civilized system, and can be implemented as sets of Prolog axioms. This makes their relationship to the DLOG data model much clearer than, say, an equivalent PL/1 implementation.

5.3.1. Embedding data sublanguages in DLOG

Despite the apparent expressive power of any data modelling technique, there eventually arises an application that strains the model to its useful limits. Often the model is simply incapable of expressing the desired information (e.g., disjunction), or is hopelessly inefficient in manipulating that information.[26] For example, the current DLOG data model does not include any explicit notion of a composite or aggregate data structure where components can be individually identified, and their relations to the larger object manipulated (cf. TAXIS [Borgida81]). DLOG does provide extensive built-in predicates for using set objects (e.g., *extension*, *member*, *subset*, *union*), so aggregate structures could be constructed by the user. Note that this does not mean that the user must construct his own sets; set objects are defined, together with many operations for manipulating them. The user's responsibility would be to construct his notion of aggregate in terms of individuals and sets.

From the efficiency viewpoint, DLOG's major shortcoming is that it expects its application database to reside in mainstore — a significantly large database would be impossible without a scheme for using secondary storage.

Traditional DBM implementations sometimes extend their flexibility by embedding the data model's manipulation primitives within a general purpose programming language [Tsichritzis77, Date81]. This notion is sometimes called an embedded data sublanguage (DSL) or a "self-contained data language."[27] Similar notions have been used in AI representation systems (E.g., MICRO-PLANNER [Sussman71], QA4 [Rulifson72]), and have usually referred to this extra flexibility as "procedural attachment."

[26] In Artificial Intelligence these notions have been called, respectively, *epistemological* and *heuristic* adequacy [McCarthy69, McCarthy77].

[27] E.g., see [Tsichritzis77, p. 59].

The DLOG data sublanguage includes the DLOG meta predicates (e.g., *derivable*, *satisfies*), and the rest of the data sublanguage is embedded with *no further effort*. The designer of an application database may specify Prolog programs, if necessary, together with the existing DLOG implementation predicates. Furthermore, user-written manipulation routines can be monitored to ensure database integrity. Because all user-asserted axioms are read and parsed by DLOG, each predicate can be compared with a restricted set provided for user use. DLOG maintains an internal data dictionary of all user-invokable DLOG primitives, and can compare the predicates used in new assertions to this set.

The current DLOG implementation includes a mechanism for loading and invoking user-defined commands, implemented in the DLOG defined Prolog subset. The command processor (§5.4.4.1) will try to interpret any unknown input as a user command. To define a user command, the user simply provides the command's data sublanguage axiomatization, and includes the command name in the user data dictionary by using the predicate *user − command*.

The *transcript* user command of the Department Database (§2.3.1) is a good example of an application-dependent user command. When invoked, the command provides the user with a menu of the following alternatives:

```
1: load
2: save
3: list
4: edit
5: create
6: browse

Selection?
```

Because student transcripts are the most volatile portion of the DDB, a facility for their separate manipulation would be more efficient than considering the whole DDB at once. As mentioned in §2.3.1, DLOG views the application database as a single theory, but *transcript* is a user-defined facility that makes more efficient use of DLOG by providing its own view of the data. The DDB transcript facility has its own definition of what a transcript is (cf. "subschema" or "frame"), and uses this definition in manipulating the DDB. For example, the transcript list option (3) is defined by an axiom like this:

```
list-transcript(*x) <-
                LIST(age-of(*s,*a))
                LIST(registered(*s,*p,*y))
                LIST(completed(*s,*c))
                LIST(grade-of(*s,*c,*g))
                LIST(course-enrolled(*s,*c))
```

Just as one might identify a particular transcript subschema or frame by its individual name in DBM or AI systems, providing *list-transcript* with a particular student name identifies the assertions that comprise that student's transcript.

This transcript facility is coupled to the DLOG implementation in a domain-independent way by including the assertion

```
user-command(transcript)
```

in the application database's data dictionary.

5.3.2. Integrity maintenance

Integrity constraints are really a specification of what can be done to a database without destroying its logical consistency. DBM research calls this "semantic integrity," to distinguish it from physical integrity. According to LaFue [LaFue82] there are very few implementations of systems with automatic integrity enforcement. DLOG's integrity processor compares favourably with the most sophisticated DBM implementation (see [Stonebraker75, Woodfill81]).

Two major classes of constraints have been identified in the database management and logic database literature: constraints that verify the consistency of a new assertion before accepting it as an axiom are called "state constraints" by Nicolas and Gallaire [Nicolas78b]. These are distinct from constraints that not only accept new assertions, but remove existing ones: these are called "transition constraints," and are a subject of lengthy discussion in the description of the TAXIS system [Mylopoulos80]. The application of state constraints has a strong theoretical foundation (e.g., see [Reiter81, Reiter83]), but transition constraints are more difficult because they assert properties on different databases. The current implementation of DLOG can manage both kinds of constraints, however, only state constraints are specified in the DLOG data model specification. The semantics of transition constraints requires some kind of formalization of operations that not only augment a database with a new update, but potentially remove existing assertions to maintain the consistency of a database. As suggested in §3.6, the meaning of transition constraints could be brought within DLOG's logical foundation by adopting a more elaborate representation. Even if this were to be done, the extended representation would not solve the practical problem of automatically maintaining multiple versions of a changing world. The problem of providing an efficient and logical solution to the problem requires a specification of the semantics of destructive assignment.[28] DLOG does not provide such a theory, but does use destructive assignment to model change in the world being described. The DLOG implementation provides transition constraints by providing user access to a deletion predicate. For example, the DDB includes the constraint:

```
completed(*s,*c) ->
          course-enrolled(*s,*c)
        & grade-of(*s,*c,*g)
        & passing-grade-of(*c,*p)
        & greater-or-equal(*g,*p)
        & DELETE(course-enrolled(*s,*c))
```

The constraint asserts that if you complete a course by attaining a passing grade, then you are no longer enrolled in that course. The semantic difficulty is that the satisfaction of this constraint requires an interpretation over domains consisting of individual databases. The practical difficulty is that the user must interpret the meaning of a transition constraint operationally, instead of using the DLOG semantics specified in Chapter 3.

As previously claimed, DLOG constraints provide a straightforward way of defining relation schemas:

[28] R.A. Kowalski, invited address "Problems of logic programming," to the Second International Conference on Logic Programming, Tuesday July 3, 1984, Uppsala University, Sweden. suggested that the semantics of destructive assignment is a serious problem for all of computing.

```
head−of(*h,*d) ->
            faculty−member(*h)
         & department(*d)
```

This constraint specifies the attributes of the relation *head−of*, i.e., constrains the argument types of the predicate *head−of*. The constraint

```
course−enrolled(*s,*c) ->
                 course(*c)
               & (registered(*s,*p,*) | eligible−for−admission(*s,*p))
               & satisfies(*s,set(*x:course−prerequisite(*c,*x)))
               & student−program−contribution(*s,*c)
```

says that "you can enroll in a course if you're registered (or eligible to register), have satisfied the course's prerequisites, and the course is a contribution to your program."

5.3.3. Transaction processing

The concept of a transaction has been used extensively in DBM [Gray81], usually as a mechanism to enforce database integrity on a batch of updates. For those familiar with the nomenclature of Badal and Popek [Badal79], DLOG uses "post execution semantic integrity validation" within the transaction processor. In other words, integrity constraints are applied after the transaction completes, and all assertions of the transaction are retracted if any constraint fails.

5.3.4. Data dictionaries

In DBM, a data dictionary provides information *about* the database — it contains meta level information. For example, IBM's System R has a data dictionary that includes relations that describe the *TABLE*s and *COLUMN*s of an application database.[29] System R's data dictionary is organized in relations just like the rest of the application database, and can be interrogated with the same query mechanism.

A similar mechanism is provided in DLOG. For example, the *TABLE* relation of System R lists

[29] See [Date81, p. 138].

all database relations and the number of columns in each, viz.

TABLE

RELATION	COLUMNS
course	1
course-no	2
.	.
.	.
.	.

The DLOG data dictionary uses *user−predicate* to achieve the same effect:

$user-predicate(course,1)$

$user-predicate(course-no,2)$

.
.
.

This information can be examined with the browse mechanism described in §5.2.4.5, or used by the system to reject queries with unknown predicate names.

Since DLOG includes rules as well as facts within its language, there is no reason to exclude rules from the data dictionary — the underlying inference mechanism can make inferences in both the object level and meta level database with equal ease. For example, by classifying the user predicates by topic and storing that classification in the data dictionary, we provide a facility for browsing the user predicates by topic (see §5.4.4.5). The DDB application includes the following topics in its data dictionary:

$topic(advising)$

$topic(registration)$

$topic(courses)$

$topic(standing)$

$topic(admission)$

$topic(grades)$

$topic(promotion)$

Each topic is related to a user predicate by the data dictionary predicate *topic−category*, e.g.,

$topic-category(admission,program-prereq)$

$topic-category(admission,faculty-program-prereq)$

topic−category(admission,dept−program−prereq)

.
.
.

Furthermore, topics can be structured into a hierarchy with the *sub−topic* predicate, e.g.,

subtopic(advising,registration)
subtopic(advising,courses)

.
.
.

Finally, because we can interpret the data dictionary like any other database, general relationships between topic categories and subtopics can be specified:

```
topic−category(*t,*c) <-
            subtopic(*t,*st)
          & topic−category(*st,*c)
```

This topic information can be used to retrieve all user predicates relevant to a particular topic — this is what the topic browser does (see §5.4.4.5). Notice that this facility is directed at users who are unfamiliar with a particular application database. Since we cannot assume that a user is familiar with topic names, this DLOG implementation includes a topic synonym predicate to help map user topic names to their data dictionary equivalents. For example, the data dictionary assertions

topic−equivalent(advising,advice)
topic−equivalent(advising,counsellor)
topic−equivalent(advising,counselling)
topic−equivalent(advising,counsel)

will allow predicates relevant to *advising* be retrieved by using those synonyms.

5.4. Heuristic interpretation of queries

Despite claims that the issues of maintaining soundness and completeness are merely artifacts of the logical paradigm, a certain freedom can be had by viewing computation as inference. The idea is that the object level behaviour of a theorem-proving algorithm (or procedure) can be controlled by a theorem-prover that deals with properties and relations on the proofs view as objects. This idea is usually attributed to the GOLUX project of Hayes [Hayes73] and has since been exploited in some actual implementations (e.g., [Bundy81, Kleer77]).

One thing that critics of the logical paradigm (e.g., [Minsky75) have overlooked is that there is no "regulation" that requires that the behaviour of a complex logic-based reasoning system be based only the notion of a *completed* proof. The heuristic method described below uses controlling axioms at the meta level to make use of partial proofs as assumptions in the query evaluation process.

5.4.1. Partial matching and partial proofs

Before describing the matching heuristic embedded in DLOG, we review two related notions of matching. The first comes from Hayes-Roth [Hayes-Roth78] who discusses one view of partial matches and their general use within so-called "pattern-directed inference systems." The second view is that of the match framework of KRL [Bobrow77b, Bobrow77a]. It is also worth noting that both of these discussions cite the matching component of MERLIN [Moore74] as background. We briefly explain MERLIN's reasoning technique before turning to a discussion of Hayes-Roth and the KRL system.

The MERLIN system sought to produce a model of analogical reasoning by employing a matching procedure called "mapping" to data structures call "β-structures." A β-structure consists of a concept name, a class name, and a list of properties or features which are themselves β-structures. A mapping operation proceeds by "positing" one β-structure as another, and then recursively making "sub-posits" between corresponding properties of the original "posit." For example, a β-structure for the concept *man* might be something like

MAN [MAMMAL NOSE [...] HOUSE [...] ...]

where MAN is the concept name, MAMMAL the class name, and the remaining properties are β-structures describing various properties of the MAN concept. Similary, the concept *pig* might be rendered as

PIG [MAMMAL SNOUT [...] STY [...] ...]

A comparison of MAN and PIG would be performed by a mapping of the MAN β-structure onto the PIG β-structure, producing the posit "MAN/PIG." After one level of sub-positing, the mapping could report that a "MAN is like a PIG if a NOSE is like a snout and a HOUSE is like a STY." This mapping process is touted as a basis for analogical or metaphorical reasoning, however the β-structures have no clearly defined meaning, and the mapping procedure is not well defined.

Despite these problems, the notion of partial match by this recursive sub-positing has been acknowledged as an important contribution to symbolic reasoning by both Hayes-Roth and Bobrow and Winograd.

Hayes-Roth's [Hayes-Roth78] discussion of partial matching is rather vague, but his general framework focuses on the results of an attempted matching of two "descriptions" A and B, described as the three-tuple $(A^*B, A-A^*B, B-A^*B)$. A description is apparently any symbolic data structure. The A^*B component denotes what A and B have in common. The two remaining components denote the "residue" of A and B, or what of A and B remain after extracting their common parts. The only thing of further relevance here (aside from this nomenclature) is Hayes-Roth's suggestions for possible uses of partial matches. He explains how the mapping of MERLIN simulates "analogical" reasoning by using partial instead of complete matching. The important concept for DLOG is that a partial match is evidence enough to continue the reasoning in progress.

Bobrow and Winograd's description of KRL's matching framework (see [Bobrow77b, §2.5]) does not focus exclusively on the notion of partial match, but their discussion about what is deductive and what is heuristic is sufficiently interesting to pursue here, because DLOG already provides some of the features of KRL's "flexible" matching.

The basic data type of KRL is a frame-like structure called a "unit." A unit is a collection of "descriptors" that attribute various properties to the unit in which they appear. Appendix 3 contains examples of KRL's various data structures. Of interest here are the various ways in which units can be related by matching their descriptors. For example, consider KRL's matching by "using properties of the datum elements" [Bobrow77b, pps. 23-24]:

> Consider matching the pattern descriptor *(which Owns (a Dog))* against a datum which explicitly includes a descriptor *(which Owns Pluto)*. The SELF description in the memory unit for Pluto contains a perspective indicating that he is a dog. In a semantic sense, the

match should succeed. It can only do so by further reference to the information about Pluto.

This form of matching can be simulated in DLOG. For example, the KRL descriptors *(which Owns (a Dog))* and *(which Owns Pluto)* might be rendered as

$$Owns(^{*}x, an(^{*}y, dog(^{*}y)))$$

and

$$Owns(^{*}x, Pluto)$$

It seems that such descriptors in KRL always appear within the scope of a particular unit, so that they attribute a property to the individual described by that unit. In this case the variable $^{*}x$ above would be interpreted as an existential variable, and we anticipate its use as a referent to an individual with the property of owning a dog. If we have the fact that Pluto is a dog (i.e., the assertion $dog(Pluto)$), then DLOG unification rule 6 (§5.2.3.1) will successfully unify the above pair by recursively proving that $dog(Pluto)$ follows from the database.

Several other forms of matching fall into similar categories, where a recursive proof will provide the inferences required to demonstrate the equality of descriptions. The only clear instance in which partial matches arise are due to resource limitations. Again the partial results determine whether the current line of reasoning is to continue (perhaps given further resources), or to be abandoned.

From both of the above views, one important use of partial matching is to sanction continuation of a current line of reasoning. In DLOG, a "line of reasoning" is simply the current search for a proof of a DLOG query. The DLOG partial match heuristic is a way of continuing the search by recording a partial match of two terms as an assumption under which the current search for a proof can continue. The relatively vague notion of partial match described above is articulated in DLOG as a partial proof.

5.4.2. The *extends* predicate

The heuristic in this DLOG implementation uses failure of the deductive proof procedure to initiate a heuristic derivation based on a DLOG meta predicate called *extends*. The heuristic is based on a partial match in the sense that it is applied during unification, and allows unification to succeed when it would otherwise fail. If a proof step in the current proof search fails because two descriptive terms can not be matched, the *extends* predicate can be invoked to attempt to find a partial match.

The *extends* predicate is implemented as a binary Prolog predicate that takes two DLOG sentences as arguments. The predicate is asymmetric: the first argument is the hypothesis, and the second argument is the goal. The derivation of the *extends* relation on two arguments attempts to verify that the goal follows from any part of the hypothesis. In implementation terms, *extends* reduces the hypothesis to a conjunction of atomic sentences, and attempts to show that the goal follows from some portion of the hypothesis. In other words any disjunctive symbols '|' '!' in a hypothesis are treated as conjunctions '&'; all of the following are derivable instances of the *extends* relation:

$$extends(p(^*x)\&q(^*y),p(a)).$$
$$extends(p(^*x)!q(^*y),p(a)).$$
$$extends(p(^*x)!q(^*y)|\,s(a)\&t(^*x),s(^*x)).$$

This choice of partial match definition was made because of Prolog's inability to deal with disjunctive assertions. The *extends* predicate sanctions a partial match by replacing inference with pattern-directed retrieval.

The implementation of *extends* simply takes each atom of the hypothesis and temporarily adds it to the current database. After each atom is added, an attempt is made to prove the goal from the augmented database. If the goal can be achieved before the atoms of the hypothesis are exhausted, then *extends* succeeds. Because DLOG unification can recursively invoke DLOG derivability (§5.2.3), this heuristic can involve further invocation of DLOG proofs and a proof can continue under the *extends* assumption.

To illustrate the use of this heuristic, consider the following query from the example terminal session of fig. 2.5:

<-*course−req*(*BScMajorsCS*,*first*,*lambda*(*$x,*completed*($x,$CS115$))).

This query can be paraphrased as "Is the completion of CS115 a course requirement for the first year of the BSc Majors program in CS?" The DDB assertion relevant to this query is

```
course−req(BScMajors,
           first,
           lambda(*x,completed(*x,CS115)!completed(*x,CS118))).
```

Notice that the two lambda terms are different, and in fact cannot be shown equivalent by the lambda equality schema (§3.4.2) as implemented in the DLOG unification procedure (§5.2.3, case 7): the matching subgoal

eqlambda(*lambda*(*x,—),*lambda*(*x,—))

fails, thus causing the search for a proof of the query to fail. Logically this failure is to be expected, as we are requesting a deduction of the form $(P\&\neg Q)\vee(\neg P\&Q)\vdash P$, which is not deducible. However, because $CS115$ or $CS118$ is a requirement, the query answer "no" seems somewhat unintuitive. (This is an example of what the critics of logic use as evidence against the use of the ideas of *truth* and *fact*.)

At this point, the heuristic based on the *extends* predicate can be invoked as follows:

```
<-extends(lambda(*x,completed(*x,CS115)!completed(*x,CS118)),
          lambda(*x,completed(*x,CS115))).
```

and will succeed because assuming the first disjunct of the hypothesis will easily produce a proof of the goal. At this point, DLOG unification reports the following:

```
heuristic assumption:
   completed( *x, CS115 )
   extends
   completed( *x, CS115 ) ! completed( *x, CS118 )
```

and completes the proof under this assumption. The query has not been deduced, but the user can see that $CS115$ is a suitable course requirement, in lieu of $CS118$.

5.4.3. Implementation of heuristic evaluation

The *extends* predicate is invoked from within DLOG unification by adding the following Prolog assertion to the unification definition of §5.2.3:

```
unify(lambda(*v,*e_1),lambda(*v,*e_2) <-
    heuristic-mode
  & query-mode
  & extends(*e_2,*e_1).
```

The two propositional atoms *heuristic−mode* and *query−mode* verify that DLOG is in both heuristic and query modes. The former is used because heuristic mode is very expensive — recall that the heuristic interpretation of queries is attempted only after deductive failure, so applying the heuristic in all failure cases would result in heavy overhead in most cases, and disaster in others. For example, the constraint mechanism (§§3.6, 5.2.4.2) should never use heuristic mode to complete a derivation that demonstrates the updated database is still consistent. When in heuristic-mode, this is avoided by the second atomic proposition *query−mode*.

It should be noted that the current *extends* predicate is a very limited use of the enormous potential for heuristic processing in this structure. For example, it would be relatively easy to include further heuristics based on partial matches of set terms, e.g., continue if one of two different set terms is a subset of the other.

Chapter 6
An approach to DLOG's formal semantics

6.1. Montague's concept of obligation

The semantics of DLOG's 3-sorted syntax cannot easily be specified in terms of a first-order semantics, even with special computational procedures. An appealing alternative is a second-order intensional logic as used by Montague to explain such concepts as "obligation," "event," and "task." As will be explained below, Montague's formalization of the concept of obligation [Montague74, p. 151ff.] corresponds well with the interpretation of lambda constants as regulations in the Departmental Database application domain.

Montague's system provides a way of specifying the semantics for DLOG's lambda constants. It is used here to describe the semantics of the complete DLOG language. While only DLOG lambda terms require this treatment, Montague's system provides a rather more uniform treatment of DLOG's semantics than is possible in weaker systems.

6.2. Contextual description based on a second order intensional logic

The essence of Montague's system can be explained in a relatively straightforward manner. An essential concept is the classification of individuals into categories of two different kinds. Each n place predicate constant has an associated type $<s_0, s_1, \cdots s_{n-1}>$ that indicates the kind of object that can appear in each term position: $s_i = -1$ specifies a standard[1] individual; $s_i = 0$ specifies a proposition; and $s_i \geq 1$ specifies a s_i-place predicate.[2]

[1] Here "standard individual" means the usual notion of an individual in a first order model. This is in contrast to higher-order models, where there are different kinds of individuals.

[2] See [Montague74, p. 150]. The notion of predicate used in this context is sometimes called a "relation in intension."

For example, a predicate constant P of type <-1,1> takes individual constants in its first position and unary predicates in its second. In the Department Data Base domain, the *satisfies* predicate constant has type <-1,1>, e.g., the assertion

$$satisfies(fred, \lambda x.completed(x, cs115))$$

has an individual constant 'fred' in the first argument position, and a lambda constant in the second argument position. The first denotes an individual object (the person with name 'fred'), and the second denotes a predicate specifying the property of "x completing the course CS115."

The meaning of the above assertion is assigned in a way that introduces the second and most important difference of Montague's system. The assignment of truth values to sentences is an inherently two phase process. As Montague explains [Montague74a, p. 157], an *interpretation* assigns *intensions* to symbols, and a *model* assigns *extensions*. Extensions include the standard objects well-known from traditional Tarksian first-order semantics, as well as sets of sequences of individuals. Intensions are functions from possible worlds to the universe of individuals. They are introduced in order to distinguish the sense or abstract meaning of a predicate from its denotation in a particular possible world.

Though the complexity of Montague's complete system can be perplexing, but of the complexity dissolves because of the simplicity of DLOG theories: they are finite, and the intended interpretation is over a highly restricted domain. This simplicity constrains the number of possible worlds that can serve as interpretations for DLOG theories (for example, this provides a restricted interpretation of "$\Box$"). In effect, the world that a particular DLOG database is intended to describe is *the* intended possible world for semantic interpretation. Therefore the first phase of interpreting a sentence, the selection of a possible world, is trivial.

6.2.1. DLOG's semantics in Montague's system

A possible interpretation for the DLOG language is a triple $<I,U,F>$, where I is a set of possible worlds, U is the set of possible individuals[32] and F is a function from individual and predicate constants to intensions. That is, for any individual or predicate constant c, F_c, is a function $F_c : I \rightarrow U$. This is the formal notion of intension—F_c is a function which, computed in a possible world i, produces the extensional denotation of that constant. In other words, $F_c(i)$ is the individual in possible world i denoted by the constant c.

Individual constants of DLOG's syntax correspond to possible individuals in the set of possible worlds. In any actual DLOG database, the intended possible world is fixed by that database's intended interpretation.

The set objects in the DLOG language can be expressed in several ways in Montague's system. In one method a DLOG set object is expressed as a predicate with the property of being the set in question. That is, the set constant '$\{a\}$' is semantically associated with the predicate that, in a possible world i, describes the set whose only member is the possible individual assigned to a in i. In Montague's formal language the set constant '$\{a\}$' can be expressed as [33]

$$\top Q \wedge u \Box (Q[u] \equiv u = a)$$

For example, consider the DLOG formula

$$cardinality(\{a\},1)$$

Here *cardinality* is a predicate constant with type <1,-1>; its first argument is a unary predicate constant and its second an individual constant. We can express the DLOG assertion as

$$cardinality(\top Q \wedge u \Box (Q[u] \equiv u = a),1) \tag{6.1}$$

Here Montague's '$\top$' symbol has a similar role to Russel's symbol 'ι' in the DLOG syntax. It is used to name a predicate Q, i.e., we can read '$\top Q$' as "the predicate Q such that..." The symbol '$\Box$' is needed to specify that the predicate definition is unique across all possible worlds. It asserts that the predicate Q has the same definition, regardless of in which possible world an interpretation is made.

[32] In Montague's "Pragmatics" [Montague74b] $U = \bigcup_{i \in I} A_i$, while in "Entities" [Montague74a] $U \supset \bigcup_{i \in I} A_i$, where A_i is the set of possible individuals existing in the possible world i. The interested reader is referred to those papers for a deeper understanding of the philosophical issues arising from the consideration of possible objects not included in one's selection of I, the set of possible worlds.

[33] Montague [Montague74a] uses the symbols '$\wedge$' and '$\vee$' for '$\forall$' and '$\exists$', respectively. He also uses brackets where parentheses are typical, e.g., P[x] for P(x). In addition Montague employs the symbols '$\top$' and '$\Box$', read as "the" and "necessarily," respectively. These latter symbols ~~are~~ ~~used~~ to form names of predicates.

Montague offers the following abbreviation for formula (6.1):

$$cardinality(\hat{u}u = a, 1)$$

In general, the '$\hat{u}\Phi$' syntax is shorthand for ⌜$\top Q \wedge u \Box(Q[u] \equiv \Phi)$⌝.

The above approach to expressing the meaning of DLOG sets requires that the type of a predicate constant reflect the size of the set to which it is applied. For example, we need a different cardinality predicate constant for each size set, i.e., cardinality <1,-1>, cardinality <2,-1>, etc. Since the DLOG domain is finite, this poses no theoretical problem. However, we can avoid this inelegance by interpreting a DLOG set object as an individual of Montague's logic and then rendering DLOG set descriptions as predicates that relate that "individual" to its members. In effect, we are avoiding the inelegance of the first encoding by defining a kind of local sorting within the Montague encoding of set descriptions. For example, with this strategy the set constant {a} is now written as

$$\top Q \wedge x \Box(Q[x] \equiv (Q'[x,a] \wedge \wedge y y \neq a \supset \neg Q'[x,y]))$$

or, in the abbreviated form as

$$\hat{x} Q'[x,a] \wedge \wedge y y \neq a \supset \neg Q'[x,y]$$

The new predicate constant Q' is introduced to name the relation of "being a member of set x," i.e.,

$$\forall x \forall y Q'(x,y) \equiv set(x) \wedge y \in x$$

Its introduction avoids the necessity of providing a different type for each predicate constant — for example, *cardinality* can have type <1,-1> regardless of the cardinality of the set on which it is asserted. The newly introduced predicate constant Q' avoids the need for explicitly sorting Montague's language, and permits the meaning of DLOG sets to be expressed without explicitly increasing the number of predicate constants in any particular DLOG theory. This introduction of a new "element-of" relation for each set is again theoretically feasible, and can be hidden in an implementation.

Definite sets can be similarly expressed. A definite set

$$\{x \mid \Phi(x)\}$$

is expressed as

$$\top Q \wedge x \Box(Q[x] \equiv \Phi(x))$$

or, in abbreviated form

$$\hat{x}\Phi(x)$$

There are two syntactic forms of indefinite set. The first has the general form $\{x,X{:}\Phi(x),\Omega(X)\}$; the DLOG set variable X is treated as a new individual variable, say y, and the Montague encoding is

$$\top Q \wedge y \Box(Q[y] \equiv \Omega(y) \wedge \wedge x Q\urcorner[y,x] \supset \Phi(x))$$

or the abbreviation

$$\hat{y}\Omega(y) \wedge \wedge x Q\urcorner[y,x] \supset \Phi(x))$$

Similarly the second form of indefinite set, $\{\alpha_1 c_1 \alpha_2 c_2 \cdots c_{n-1}\alpha_n\}$ is written as

$$\hat{x}(Q\urcorner[x,\alpha_1]\, c_1\, Q\urcorner[x,\alpha_2]\, c_2 \cdots c_{n-1}\, Q\urcorner[x,\alpha_n]) \wedge (\wedge y y \neq \alpha_1 \wedge y \neq \alpha_2 \wedge \cdots \wedge y \neq \alpha_n \supset \neg Q\urcorner[x,y])$$

Here the first parenthesized sub-formula specifies a set's potential members and the second sub-formula specifies its closure conditions.

Finally, DLOG lambda expressions are expressed as unary predicate constants. For example, the DLOG formula

$$requirement(BScMajorsCS, \lambda x.completed(x,CS115))$$

is written as

$$requirement(BScMajorsCS, \hat{x}completed(x,CS115)) \tag{6.2}$$

This not only shows that DLOG's lambda symbol 'λ' plays the same role as Montague's '^' symbol, but reveals a deeper equivalence: when expressed in Montague's system, the meaning of DLOG's lambda terms and set terms are identical. The reason should be apparent: it is because there is a natural relation between the notion of a set and being a member of a set. They are distinguished here so that one can refer to the set as an object. At the expense of a more complex syntax (i.e., adding set variables), we could retain the intended DLOG distinction within the Montague framework. In DLOG this distinction is made by having sorted variables.

The intended meaning of lambda expressions does not collapse in the same way as sets. This is because the formula (6.2) is intended to mean "a requirement of the BScMajorsCS program is to bear the relation *completed* to the course CS115." The intensional semantics provides a way of admitting different intensions for the *completed* relation, e.g., completing a course might have different meanings in different possible worlds. In the case of DLOG, the particular possible world in which symbols are assigned extensions is fixed to be the Departmental Database.

The second order power of Montague's logic provides the expressive ability to assert relations on predicates: it is the property of completing *CS*115 that bears the *requirement* relation to the program *BScMajorsCS*, and not any particular extension of the property. In particular, we determine the truth of formula (6.2) as follows: the predicate constant *requirement* has type <-1,1> and so denotes a relation $<I,U,U_1>$, where I is the set of possible worlds, U the set of individuals, and U_1 is the set of all predicates $<I,U>$. Now the formula (6.2) is true in a structure $<I,U,F>$ just in case there is a denotation $x \in U$ for *BScMajorsCS* and $A \in <I,U>$ for $\hat{x}completed(x,CS115)$ such that $F_{requirement}(i)$ holds on $<x,A>$. (Recall that $F_{requirement}$ is the intension of the predicate constant *requirement*, and that $F_{requirement}(i)$ is the extension of *requirement* in possible world i.)

6.2.2. Interpretation of DLOG based on intensional logic

In general, the interpretation of Montague's formulas is specified as follows. This specification is adapted from Montague [Montague74a, pps. 150-159]. Let $<I,U,F>$ be a possible interpretation, and let i be a member of I.

(1) If u is an individual variable and c an individual constant, then the possible individual x satisfies the formula $u=c$ (in i with respect to $<I,U,F>$) if and only if x is identical with $F_c(i)$.

(2) If in addition, P is a predicate constant of type <-1,-1>, then x satisfies the formula $P[u,c]$ (in i, with respect to $<I,U,F>$) if and only if the pair $<x,F_c(i)>$ is a member of $F_P(i)$.

(3) If u is an individual variable, Q a one-place predicate variable, Φ a predicate constant of type <-1,1>, x a member of U, and A a predicate of type $<I,U>$, then the pair $<x,A>$ satisfies the formula $\Phi[u,Q]$ (in i, with respect to $<I,U,F>$) if and only if $<x,A>$ is a member of $F_\Phi(i)$.

(4) If P is a zero-place predicate variable and A a predicate of type $<I>$ (that is, a proposition),

then A satisfies the formula $P[\,]$ (in i, with respect to $<I,U,F>$) if and only if the empty sequence is a member of $A(i)$ (that is, if and and only if $A(i)$ is truth).

(5) If Φ is a sentence, then $\neg\Phi$ is true (in i, with respect to $<I,U,F>$) if and only if Φ is not true (in i, with respect to $<I,U,F>$); similarly for other sentential connectives.

(6) If u is an individual variable and Φ a formula of which u is the only free variable, then $\bigvee u\Phi$ is true (in i, with respect to $<I,U,F>$) if and only if there is an object x in U such that x satisfies Φ (in i, with respect to $<I,U,F>$).

(7) If Q is an n-place predicate variable and Φ a formula of which Q is the only free variable, then $\bigvee Q\Phi$ is true (in i, with respect to $<I,U,F>$) if and only if there is a predicate of type $<I,U_0,...,U_{n-1}>$ which satisfies Φ (in i, with respect to $<I,U,F>$), where each U_k (for $k<n$) is U.

(8) If Φ is a sentence then $\Box\Phi$ is true in i (with respect to $<I,U,F>$) if and only if Φ is true in j (with respect to $<I,U,F>$), for every j in I.

(9) If u is an individual variable, Q a one-place predicate variable, Ω a predicate constant of type $<$-1,1$>$, Φ a formula of which the only free variable is Q, and x a member of U, then x satisfies $\Omega[u,\top Q\Phi]$ (in i, with respect to $<I,U,F>$) if and only if $<x,A>$ is in $F_\Omega(i)$, where either

   (i) there is exactly one predicate of type $<I,U>$ which satisfies Φ (in i, with respect to $<I,U,F>$), and A is that predicate, or

   (ii) it is not the case that there is exactly one such predicate and A is the empty predicate (i.e., $I\times\{\Lambda\}$).

Finally, we require an addition that considers the meaning of DLOG's definite individuals. While Montague's language uses the symbol '$\top$' to form new predicate constants, the language has no equivalent symbol for forming individual names. Therefore we add

(10) $P(\iota x\Phi(x))$ is true (in i with respect to $<I,U,F>$), if and only if either

   (i) there is exactly one u in U such that u satisfies $\Phi(x)$ (in i with respect to $<I,U,F>$) and

$u \in F_P(i)$, or

(ii) there is not exactly one u such that u satisfies $\Phi(x)$ (in i with respect to $<I,U,F>$) and there is a distinguished individual $\Lambda \in F_P(i)$.

The special individual Λ is the individual, different from any other individual, that is denoted by those definite individuals that are ambiguous. In the DLOG implementation each definite individual is subject to reference determination. Definite individuals that fail to refer cause the rejection of the assertion in which they appear. In theory, the selected individual is that special individual that achieves parsimony in the specification of the first order semantics of definite descriptions (cf. [Rosser68, Kaplan75, p. 215]). In application, Λ represents the computer program that rejects the offending assertion (for further details see §4.3.1).

To conclude this section, we note that Hilbert's symbol 'ϵ' has not yet been given a meaning within Montague's system. This is because we desire an interpretation where 'ϵ' is treated as a selection operator(cf. [Leisenring69, p. 4-5]). The particular individual denoted by a term of the form '$\epsilon x.\Phi(x)$' is determined proof theoretically — or in implementation terms, by the index structure of a particular DLOG database implementation (see §§4.2, 4.3.2, 5.2.3.1).

6.3. Higher order intensional proof theory?

While some basis exists for the implementation of higher-order proof theories based on extended unification (e.g., [Huet75, Jensen75, Darlington77]), no such basis exists for intensional higher-order logic. Part of the reason for the lack of this foundation is evident in the following quote taken from an introductory book on Montague's semantic theory:

> "...we will concentrate exclusively on model theoretic definitions of semantic entailment, validity and related notions, rather than deductive systems. This is not to say that the study of deductive systems has *no* interest for semantics and pragmatics of natural language. It might, for example, have particular applications in the psycholinguistic study of how people draw inferences from sets of sentences, or in artificial intelligence studies. Rather, this means that we can safely ignore formal deductive systems...since our model-theoretic method renders them superfluous for our purposes."[34]

[34] [Dowty81, p. 53].

As the development and application of Montague's intensional logic was not motivated by deductive concerns, it is not surprising to find that little has been done to consider the nature of an intensional higher-order proof theory. As computational linguists develop a coherent model of language interpretation based on Montague semantics, the need for a computational understanding will grow. For example, Hobbs and Rosenschein have already illustrated the relationship between Montague's intensional interpretation of quantification and LISP [Hobbs78].

An essential notion in any intensional proof theory is the definition of a syntactic operator that performs the equivalent of assigning an intensional to a syntactic object. This is the first phase of truth determination in an intensional logic, and a suitable proof-theoretic correlate is required.

As an example, consider the intensional interpretation of DLOG's lambda constants. As intensional objects, their proof-theoretic manipulation should require a mechanism that considered something like a set of possible theories as the corresponding syntactic notion of possible worlds. Of course the interpretation of the DLOG lambda constant is simplified by having only *one* possible intended interpretation. In that situation, lambda constants can be manipulated as if their intension had been determined.

As others have indicated, there are many natural language utterances that require an intensional interpretation because of the opaque contexts created by propositional attitudes. These cases do not arise in DLOG, and so the derivation strategy specified in §3.4 is sufficient. It seems, however, that the general problem of how to provide an effective proof-theoretic equivalent of constrained subsets of higher-order intensional logic is an open question.

Chapter 7
Conclusions

The motivation for the specification and construction of DLOG shares goals from Database Management (DBM) and Artificial Intelligence (AI). The former emphasizes data independence and the efficient organization and use of large volumes of information, while the latter emphasizes conceptual efficiency in representing information and the recovery of implicit information by drawing inferences. The DLOG system combines these motivations, using logic as a unifying methodology.

DLOG demonstrates that elaborate descriptive terms can be used to simplify the task of describing an application domain. DLOG further demonstrates that such terms can be embedded in a Horn clause syntax and effectively manipulated within the SLD resolution theorem-proving framework. Both the specification and implementation demonstrate the usefulness of decoupling a theorem-prover's equality theory from the general proof procedure. This decoupling allows non-standard syntactic objects like lambda terms to be cleanly manipulated with meta language predicates.

The methodology employed in the design, specification, implementation, and application of DLOG provides an example of how logic can be used to achieve a symbiosis of database management and knowledge representation ideas. DLOG is unlike any traditional database management system, yet it usefully incorporates many of their features in a working system that has been applied to a real domain.

7.1. Contributions as traditional Database Management

The prototype DLOG system provides a database management system that provides traditional facilities like integrity maintenance, data dictionaries, transactions, and a database sublanguage. In addition, the standard facilities are integrated with a deductive mechanism so that every operation can be explained in terms of deduction. The DLOG implementation demonstrates that important database management concepts are realizable in terms of deduction based on logic programming.

DLOG's data description or "representation" language provides rich language for expressing information in a data independent way. DLOG's descriptions of individuals, sets, and lambda expressions are more powerful than any current traditional data description language. The transformation of these complex syntactic entities to simpler ones is dynamic in DLOG; the proof-theoretic definition of equality provides the mechanism, and the method of contextual definition provides the meaning. In other words, part of the DLOG language's richness comes from not forcing the user to "normalize" his input; the underlying deductive mechanism is used to provide standard interpretations of the more elaborate input language.

Integrity maintenance is provided by interpreting a database as a restricted logical theory, and by defining integrity as a form of deductively-determined assertability. The generally undecidable problem of proving consistency after an update is restricted by the simple assumption that integrity can be defined as demonstrating a subset of integrity constraints re-provable. update. Instead of testing arbitrary theories for consistency, we incrementally test the consistency of an update against the finite database and a restricted class of constraints relevant to that update. By defining a syntactic class called "integrity constraint," deduction can be used to verify the consistency of an update. The algorithm for this style of integrity maintenance is as powerful as any currently existing database system, and is simple and straightforward from a logic programming perspective.

The logic programming implementation of a data dictionary shows that no special mechanism is necessary to provide a user with facilities for asserting relations on relations. The DLOG data dictionary is simply a subset of the complete DLOG database, and can be manipulated with the same deductive mechanisms.

The uniformity of DLOG provides a powerful datasublanguage in a simple way. There is no need for a complex software interface to define the relationship between a foreign data sublanguage and the data definition language (e.g., for using PL/1 to manipulate databases). The Prolog definition language of DLOG is simply a subset of the DLOG language, and user-dependent requirements that cannot be met by DLOG can be easily provided in Prolog (e.g., see §5.3.1).

While DLOG demonstrates the advantages of using a logic programming approach to database management definition and implementation, there are several features lacking in the current prototype. These include facilities that are most important in actual database management systems, but for which we have no implementation. Included here are things like software for concurrency control, efficient (indeed, any) use of secondary storage, error recovery mechanisms, security authorization, and data communication services. Some of these facilities will be forthcoming with more sophisticated logic programming systems (e.g., ESP [Chikayama83]), but others are lacking simply because their provision was not considered important in the prototype development (e.g., error recovery).

7.2. Contributions as Artifical Intelligence

DLOG has been implemented and applied to a real domain. The DLOG data model and experimental implementation represents another step in the direction initiated by Hayes' concept of a *representation scheme* [Hayes74]. The original goal of using the data model concept to guide the design and construction of a "knowledge base management system" is fully realized in DLOG. Although DBM researchers do not agree on the components that constitute a data model (see §2.2), a reasonable approximation results in a clear separation of

(1) data definition (representation) language,

(2) semantic theory, and

(3) relationships between assertions, queries, and application databases (theories).

Hayes' representation scheme concept encompasses (1) and (2), and is well met by adopting a logical foundation. In AI, the transition from scheme to system requires a specification of the details for (3), and this has typically been the area where semantics and implementation become entangled and confused. AI formalists have alway argued that representation languages, *not* systems, must be given denotational semantics independent of implementation [Hayes77, McDermott78b]. The data model concept provides a framework for pursuing this goal: *data independence* is the goal behind the adage "no notation with denotation" [McDermott78b].

The DLOG representation language achieves implementation independence by using the data model concept. An implementation could have been done in LISP or PASCAL *without altering the intended meaning of DLOG sentences.* The DLOG system is an implementation of a representation language whose meaning is independent of implementation details.

The inadequacies of any general purpose data description language usually require a supporting data manipulation language to capture the idiosyncracies of application databases. This view, gleaned from DBM, offers a new view of "procedural attachment" in AI languages. The *user_command* facility in DLOG provides for procedural attachment in a way that is independent of application domain. Furthermore, the DLOG system's data dictionaries provide the mechanisms for monitoring all user-defined data sublanguage programs to avoid the creation of objects not interpretable by the DLOG data model's semantic theory.

Although the DLOG language does not include any explicit notion of frame, script, schema, plane, or depiction, it does include a technical contribution to representation languages. Many have acclaimed the advantage of "the ability to construct partial descriptions of an object without identifying the object,"[35] but there have been few attempts to use logic as the basis for doing so: the DLOG system contains the most comprehensive set of embedded terms of any system known to this author.

The implementation of DLOG has produced two contributions to knowledge representation techniques. First, this implementation shows the impact of admitting descriptions as syntactic objects (§4.2). The DLOG interpretation of descriptive terms is but a scratch on the surface of an old philosophical problem, but, by using a logical approach, the problems are at least clarified. Ad hoc approaches to the treatment of descriptions are in danger of stumbling on, or even ignoring the contributions of philosophers to a theory of descriptions and naming [Carroll78].

The second technical contribution of DLOG's term interpretation is the way in which the deduc-

[35] [Robinson80, p. 150].

tions required to match DLOG terms are initiated via the "unification" procedure.[36] This mechanism is an extension of Reiter's idea [Reiter75], who observed that any deductive abilities that could be effectively offloaded to the matching component of a theorem-proving procedure would improve its efficiency. DLOG's interpretation of embedded terms exploits this idea, and provides an initial analysis of the mapping-based reasoning proposed in KRL. DLOG further illustrates that this embedded style of descriptions can use a logical proof theory as the basis for plausible reasoning. The *extends* predicate (§5.4) is one small part of a notion of plausible reasoning, but it shows that the perception of logic-based reasoning as too rigid (or "too all-or-nothing," or "too zero-one") is unfounded. Even the simple DLOG heuristic based on partial proof shows that the logical methodology can support plausible reasoning strategies.

7.3. Prolog as an implementation language

The advantage of using Prolog as an implementation language is evident in the statement of an AI researcher outside the logic programming community:

> ...Prolog supplies certain AI-oriented features, such as pattern-matching and an assertional database, so that the user doesn't have to provide them...Two groups of students were taught AI programming, one in POP-2 (a LISP-like language), and one in Prolog. After two months, the POP-2 group was writing pattern matchers, and the Prolog group was writing natural language question answerers.[37]

A principle drawback of Prolog is that there are no production quality implementations (e.g., see [Moss80]), and only limited support facilities. Arguments for adapting the elaborate environments of LISP (e.g., [Komorowski81, Robinson82a]) are not yet conclusive. In addition, it is not only good editing and debugging facilities that are needed, but more sophisticated database management. Efforts directed at using secondary storage have only just begun

The basic facilities of Prolog required the following extensions for the DLOG implementation: explicit requests for deterministic derivation, creation and destruction of local proof contexts,

[36] The quotes concede that DLOG unification is based on, but not equivalent to Robinson's original definition [Robinson65].

[37] [McDermott80, p. 18].

retractable database updates, and proof procedure extensions for various kinds of disjunction and equivalence. This should be enough evidence to show that Prolog is but *one* kind of logic programming, and that heavy investments in lieu of investigating alternatives may not be warranted (e.g., [Bowen80, Bruynooghe82, Naish83b, ICOT83]). In many ways, the criticism of MICRO-PLANNER as a programming language [Sussman72] should be taken to heart once again by those prone to equate logic programming and Prolog programming.

7.4. Future research

The ideas developed in DLOG could be further pursued along several avenues, some of which reflect the merging of AI and DBM research developments. For example, the development of expert systems in AI could benefit from DLOG-like implementations. The claim that embedded descriptions provide some kind of conceptual ease of expression could further be verified by using them in expert development systems like EMYCIN [Melle81]. Most of the effort required will be in implementing knowledge acquisition tools for use with the DLOG data model (e.g., [Davis76]).

The use of descriptions in a formal language unearths a spate of philosophical problems related to the status of names and naming (e.g., [Donnellan66, Brinton77, Katz77]). The KRL language and DLOG's interpretation based on logic provide evidence that AI studies of naming and descriptions in limited domains may help shed light on their computational aspects (this is anticipated by Carroll [Carroll78]). This is clearly true in philosophy, where, for example, Katz' analysis of descriptions and naming exploits computational concepts to describe naming phenomena.[38]

As discussed in §§2.3.1, 3.2.3, 5.3.2, DLOG provides no semantic foundation for dealing with the maintenance of a representation of an evolving world. The problem of providing this foundation as well as a computationally practical implementation of it remains unsolved. One possible idea, as regards semantics, is to investigate the maintenance of evolving databases based on possible worlds semantics (cf. §6.2). For example, the current implementation of DLOG admits only one of the

[38] See [Katz77, p. 47].

possible worlds of the Montague possible world semantics (§6.2). By admitting and maintaining multiple theories whose intended interpretations corresponds to time related possible worlds, we may get a reasonable semantic foundation within the current framework. Whether this will provide insight into a computationally practical maintenance algorithm remains to be determined. One idea here is to classify all DLOG application relations as timeless or not timeless, and then develop a data structure that provides efficient access to a modified DLOG proof procedure that manages time. Such a scheme is currently under investigation.

The idea of using partial proofs to generate assumptions that sanction plausible query answers deserves much more attention. Future investigations should include classification of different kinds of plausible reasoning that can be modelled with such an approach, as well as empirical investigations directed at improving the efficiency of the failure-directed invocation of heuristics. Note that much of the inefficiency in the Prolog style of derivation stems from unnecessary backtracking; heuristics that are applied only after deduction fails must await exhaustion of all derivation alternatives. Even with proof procedures effective for finite domains, the inefficiency is potentially outrageous.

Further in this vein is the possibility of developing a reasoning system that uses multiple theories of equality; van Emden and Lloyd's [Emden84] general SLD proof procedure might be augmented with any number of equality theories, each with a special purpose. The *extends* heuristic of DLOG is an alternative form of equality, invoked when "standard" equality proofs fail. It will be interesting to consider multiple equality theories that range in their fidelity to logical deduction, e.g., "strictly equal," "similar," "nearly identical," etc.

A related pursuit is to investigate the relationship between DLOG unification and Eqlog's narrowing procedure. More generally, one would like to be able to specify both the logical and computational properties of a family of equality theories in order to be able to understand the affects of augmenting an elementary logic with various kinds of equality axioms.

There is, of course, some future in developing DLOG ideas for traditional DBM. At least two prospects seem interesting. First, much of the power of DLOG comes from using a proof theory as a computational mechanism to manipulate the "intensional"[39] component of a relational database system. DLOG shows that not only object level inferences (e.g., for evaluating queries against a database of general facts) are useful, but that many DBM facilities can be characterized at the meta level as the manipulation of theories (cf. [Kowalski81]). Further work in this area would involve experimental implementations of the ideas similar to Mylopoulos et al. [Mylopoulos80, Borgida81], with a commitment to investigating database states as objects of logic-programming theories.

A second prospect is to investigate the feasibility of combining a DLOG-like front end with an existing DBM system (e.g., INGRES). Like Reiter's proposal for a deductive question-answering facility [Reiter78a], the DBM component would provide facilities for managing large extensional databases, and the DLOG front end would provide intelligent user interface facilities based on logic programming.

The ambitious fifth generation project of the Japanese [Fuchi81, ICOT82, ICOT83, Warren82, Kowalski82, Moto-oka83] deserves notice in this regard, since it may be plausible to believe that future machines will be logic programming machines whose speed is measured in logical inferences per second (LIPS) instead of the now traditional floating point operations per second (FLOPS). Should these developments be successful, the application of DLOG-like systems will consolidate AI and DBM to a stronger degree than anyone can yet predict.

[39] Logic database theorists have used the terms "extensional" and "intensional" to describe, respectively, the tables and general constraints of traditional DBM systems (e.g., [Reiter78a]).

References

[Abrial74] J.R. Abrial (1974), Data semantics, *Data Management Systems*, J.W. Klimbie and K.L. Koffeman (eds.), North Holland, Amsterdam, 1-60.

[Alps81] R.A. Alps and R.C. Neveln (1981), A predicate logic based on indefinite description and two notions of identity, *Notre Dame Journal of Formal Logic* **22**(3), 251-263.

[Armstrong80] W.W. Armstrong and C. Delobel (1980), Decompositions and functional dependencies in relations, *ACM Transactions on Database Systems* **5**(4), 404-430.

[Astrahan76] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson (1976), System R:relation approach to database management, *ACM Transactions on Database Management* **1**(2), 97-137.

[Attardi81] G. Attardi and M. Simi (1981), Consistency and completeness of Omega, a logic for knowledge representation, *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, August 24-28, The University of British Columbia, Vancouver, British Columbia, 504-510.

[Badal79] D.Z. Badal and G.J. Popek (1979), Cost and performance analysis of semantic integrity validation methods, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 30-June 1, Boston, Massachusetts, 109-115.

[Berstein76] P.A. Berstein (1976), Synthesizing third normal form relations from functional dependencies, *ACM Transactions on Database Systems* **1**(4), 277-298.

[Bledsoe77a] W.W. Bledsoe (1977), Set variables, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, August 22-25, Cambridge, Massachusetts, 501-510.

[Bledsoe77b] W.W. Bledsoe (1977), Non-resolution theorem proving, *Artificial Intelligence* **9**(1), 1-35.

[Bobrow77a] D.G. Bobrow and T. Winograd (1977), Experience with KRL-0, one cycle of a knowledge representation language, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, August 22-25, MIT, Cambridge, Massachusetts, 213-222.

[Bobrow77b] D.G. Bobrow and T. Winograd (1977), An overview of KRL-0, a knowledge representation language, *Cognitive Science* **1**(1), 3-46.

[Bobrow79] D.G Bobrow and T. Winograd (1979), KRL, another perspective, *Cognitive Science* **3**(1), 29-42.

[Borgida81] A. Borgida and H.K.T. Wong (1981), Data models and data manipulation languages: complementary semantics and proof theory, *Proceedings of the Seventh International Conference on Very Large Data Bases*, September 9-11, Cannes, France, 260-271.

[Bowen80] K. Bowen (1980), Logic programming and relational data bases progress report, *Proceedings of the Logic Programming Workshop*, July 14-16, Debrecen, Hungary, S.-A. Tarnlund (ed.), 219-223.

[Bowen82] K. Bowen and R.A. Kowalski (1982), Amalgamating language and metalanguage in logic programming, *Logic Programming*, A.P.I.C. Studies in Data Processing 16, K.L. Clark and S.-A. Tarnlund (eds.), Academic Press, New York, 153-172.

[Brachman78] R.J. Brachman (1978), A structural paradigm for representing knowledge, Report No. 3605, Bolt Beranek and Newman, Cambridge, Massachusetts, May.

[Brachman79] R.J. Brachman, R.J. Bobrow, P.R. Cohen, J.W. Klovstad, B.L. Webber, and W.A. Woods (1979), Research in natural language understanding annual report, Report

No. 4274, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, August.

[Brachman80] R.J. Brachman and B.C. Smith (1980, eds.), Special issue on knowledge representation, *ACM SIGART Newsletter* **70**, February.

[Brinton77] A. Brinton (1977), Uses of definite descriptions and Russell's theory, *Philosophical Studies* **31**, 261-267.

[Brodie81] M.L. Brodie and S.N. Zilles (1981, eds.), *ACM SIGMOD Record* **11**, Pinegree Park, Colorado [also *ACM SIGART Newsletter* **74**, *ACM SIGPLAN Notices* **16**(1)].

[Brown78] F.M. Brown (1978), Towards the automation of set theory and its logic, *Artificial Intelligence* **10**(3), 281-316.

[Bruynooghe82] M. Bruynooghe (1982), The memory management of PROLOG implementations, *Logic Programming*, A.P.I.C. Studies in Data Processing 16, K.L. Clark and S.-A. Tarnlund (eds.), Academic Press, New York, 83-98.

[Bundy81] A. Bundy and B. Welham (1981), Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation, *Artificial Intelligence* **16**(2), 189-212.

[Carnap47] R. Carnap (1947), *Meaning and Necessity*, University of Chicago Press, Chicago, Illinois.

[Carroll78] J.M. Carroll (1978), Names and naming: an interdisciplinary view, Research Report RC7370, IBM Watson Research Center, Yorktown Heights, New York, October.

[Chamberlin81] D.D. Chamberlin, M.M. Astrahan, M.W. Blasgen, J.N. Gray, P.P. Griffiths, W.F. King, B.G. Lindsay, R.A. Lorie, J.W. Mehl, T.G. Price, F. Putzolu, P.G. Selinger, M. Schkolnick, D.R. Slutz, I.L. Traiger, B.W. Wade, and R.A. Yost (1981), A history and evaluation of System R, *ACM Communications* **24**(10), 632-646.

[Chang73] C.L. Chang and R.C.T. Lee (1973), *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York.

[Cheng84] M.H.M. Cheng (1984), The design and implementation of the Waterloo Unix Prolog environment, M.Math thesis dissertation, Department of Computer Science, University of Waterloo, September, 114 [ICR Report 26; Department of Computer Science Technical Report CS-84-47].

[Chikayama83] T. Chikayama (1983), ESP — Extended Self-contain Prolog — as a preliminary kernel language of Fifth Generation Computers, *New Generation Computing* **1**(1), 11-24.

[Clark78] K.L. Clark (1978), Negation as failure, *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 293-322.

[Clark82] K.L. Clark and S.-A. Tarnlund (1982, eds.), *Logic Programming*, A.P.I.C. Studies in Data Processing 16, Academic Press.

[Clark84] K.L. Clark and F.G. McCabe (1984, eds.), *micro-PROLOG: Programming in Logic*, Prentice-Hall, Englewood Cliffs, New Jersey.

[Clocksin81] W.F. Clocksin and C.S. Mellish (1981), *Programming in PROLOG*, Spring-Verlag, New York.

[Codd70] E.F. Codd (1970), A relational model for large shared data banks, *ACM Communications* **13**(6), 377-387.

[Codd72] E.F. Codd (1972), Relational completeness of database sublanguages, *Data Base Systems*, R. Rustin (ed.), Prentice Hall, Englewood Cliffs, New Jersey, 65-98.

[Codd74] E.F. Codd (1974), Seven steps to rendezvous with the casual user, *Data Base Management*, J.W. Klimbie and K.L Koffeman (eds.), North Holland, Amsterdam, 179-200.

[Codd79] E.F. Codd (1979), Extending the database relational model to capture more meaning, *ACM Transactions on Database Systems* **4**(4), 397-434.

[Codd81] E.F. Codd (1981), Data models in database management, *ACM SIGMOD Record* **11**(2), 112-114.

[Codd82] E.F. Codd (1982), Relational database: a practical foundation for productivity, *ACM Communications* **25**(2), 109-117.

[Colombetti78] M. Colombetti, P. Paolini, and G. Pelagatti (1978), Nondeterministic languages used for the definition of data models, *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 237-257.

[Dahl80] V. Dahl (1980), Two solutions for the negation problem, *Proceedings of the Logic Programming Workshop*, July 14-16, Debrecen, Hungary, S.-A. Tarnlund (ed.), 61-72.

[Dahl82] V. Dahl (1982), On database systems development through logic, *ACM Transactions on Database Systems* 7(1), 102-123.

[Darlington77] J.L. Darlington (1977), Improving the efficiency of higher-order unification, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, August 22-25, Cambridge, Massachusetts, 520-525.

[Date81] C.J. Date (1981), *An Introduction to Database Systems*, edition 3, The Systems Programming Series, Addison-Wesley, Reading, Massachusetts.

[Davidson82] J. Davidson (1982), Natural language access to databases: user modeling and focus, *Proceedings of the Fourth National Conference of the Canadian Society for the Computational Studies of Intelligence*, May 17-19, University of Saskatchewan, Saskatoon, Saskatchewan, 204-211.

[Davis60] M. Davis and H. Putnam (1960), A computing procedure for quantification theory, *ACM Journal* 7(3), 201-215.

[Davis76] R. Davis (1976), Applications of meta-level knowledge to the construction, maintenance, and use of large knowledge bases, STAN-CS-76-552 [also published as Stanford AI Laboratory Memo AIM-283].

[Dilger78] W. Dilger and G. Zifonun (1978), The predicate calculus-language KS as a query language, *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 377-408.

[Donnellan66] K.S. Donnellan (1966), Reference and definite descriptions, *Philosophical Review* 75(3), 281-304.

[Dowty81] D.R. Dowty, R.E. Wall, and S. Peters (1981), *Introduction to Montague semantics*, D. Reidel, Dordrecht, Holland.

[Emden78] M.H. van Emden (1978), Computation and deductive information retrieval, *Formal Description of Programming Concepts*, E. Neuhold (ed.), North Holland, New York, 421-440.

[Emden84] M.H. van Emden and J.W. Lloyd (1984), A logical reconstruction of Prolog II, *Proceedings of the Second International Logic Programming Conference*, July 2-6, Uppsala University, Uppsala, Sweden, 115-125.

[Findler79] N.V. Findler (1979, ed.), *Associative Networks: Representation and use of Knowledge by Computers*, Academic Press, New York.

[Fry76] J.P. Fry and E.H. Sibley (1976), Evolution of data-base management systems, *ACM Computing Surveys* 8(1), 7-42.

[Fuchi81] K. Fuchi (1981), Aiming for knowledge information processing systems, *Proceedings of the International Conference on Fifth Generation Computer Systems*, October 19-22, Tokyo, Japan, T. Moto-Oka (ed.), 107-120.

[Gallaire78] H. Gallaire and J. Minker (1978, eds.), *Logic and Data Bases*, Plenum Press, New York.

[Gallaire81] H. Gallaire, J. Minker, and J.M. Nicolas (1981, eds.), *Advances in Data Base Theory*, vol. 1, Plenum Press, New York.

[Gerlenter63] H. Gerlenter (1963), Realization of a geometry-theorem proving machine, *Computers and Thought*, E.A. Feigenbaum and J. Feldman (eds.), McGraw-Hill, New York, 134-152 [reprinted from Proceedings of an International Conference on Information Processing, Paris, 1959, UNESCO House, 273-282].

[Gilmore60] P.C. Gilmore (1960), A proof method for quantification theory; its justification and realization, *IBM Journal of Research and Development* 4(1), 28-35.

[Goebel77] R.G. Goebel (1977), Organizing factual knowledge in a semantic network, M.Sc. dissertation, Department of Computing Science, University of Alberta, Edmonton, Alberta, September, 99 pages.

[Goebel78] R.G. Goebel and N.J. Cercone (1978), Representing and organising factual knowledge in proposition networks, *Proceedings of the Second National Conference of the Canadian Society for the Computational Studies of Intelligence*, July 19-21, University of Toronto, Toronto, Ontario, 55-63.

[Goebel80] R.G. Goebel (1980), PROLOG/MTS User's Guide, Technical Manual TM80-2, Department of Computer Science, The University of British Columbia, December, 55 pages.

[Goguen84] J.A. Goguen and J. Meseguer (1984), Equality, types, modules and generics for logic programming, *Proceedings of the Second International Logic Programming Conference*, July 2-6, Uppsala University, Uppsala, Sweden, 115-125.

[Gray81] J. Gray (1981), The transaction concept: virtues and limitations, *Proceedings of the Seventh International Conference on Very Large Data Bases*, Cannes, France, 144-154.

[Green69] C.C. Green (1969), Theorem proving by resolution as a basis for question-answering systems, *Machine Intelligence*, vol. 4, B. Meltzer and D. Michie (ed.), American Elsevier, New York, 183-205.

[Haridi83] S. Haridi (1983), Logic programming based on a natural deduction system, Ph.D. dissertation, Department of Telecommunication and Computer Systems, The Royal Institute of Technology, Stockholm, Sweden.

[Hayes73] P.J. Hayes (1973), Computation and deduction, *Second Symposium on the Mathematical Foundations of Computer Science*, Czechoslovakia Academy of Sciences.

[Hayes74] P.J. Hayes (1974), Some problems and non-problems in representation theory, *Proceedings of the Artificial Intelligence and Simulation of Behaviour Summer Conference*, July, Univeristy of Sussex, Brighton, England, 63-79.

[Hayes77] P.J. Hayes (1977), In defence of logic, *Proceeding of the Fifth International Joint Conference on Artificial Intelligence*, August 22-25, MIT, Cambridge, Massachusetts, 559-565.

[Hayes80] P.J. Hayes (1980), The logic of frames, *Frame Conceptions and Text Understanding*, Research in Text Theory 5, Dieter Metzing (ed.), Walter de Gruyter, Berlin, Germany, 46-61.

[Hayes-Roth78] F. Hayes-Roth (1978), The role of partial and best matches in knowledge systems, *Pattern-Directed Inference Systems*, D.A. Waterman and F. Hayes-Roth (eds.), Academic Press, New York, 557-574.

[Hewitt80] C. Hewitt, G. Attardi, and M. Simi (1980), Knowledge embedding in the description system Omega, *Proceedings of the First American Association of Artificial Intelligence Conference*, August 18-21, Stanford University, Stanford, California, 157-163.

[Hilbert39] D. Hilbert and P. Bernays (1939), *Grundlagen der Mathematik*, vol. 2, Springer-Verlag, New York.

[Hirschberg84] J. Hirschberg (1984), Scalar implicature and indirect responses to yes/no questions, *Proceedings of the Fifth National Conference of the Canadian Society for the Computational Studies of Intelligence*, May 15-17, University of Western Ontario, London, Ontario, 11-15.

[Hobbs78] J. Hobbs and S.J. Rosenschein (1978), Making computational sense of Montague's intensional logic, *Artificial Intelligence* **9**(3), 287-306.

[Huet73] G.P. Huet (1973), The undecidability of unification in third order logic, *Information and Control* **22**(3), 257-267.

[Huet75] G.P. Huet (1975), A unification algorithm for typed λ-calculus, *Theoretical Computer Science* **1**(1), 27-57.

[ICOT82] ICOT (1982), Outline of research and development plans for fifth generation computer systems, Institute for New Generation Computer Technology, Tokyo, Japan, May.

[ICOT83] ICOT (1983), Outline of research and development plans for fifth generation computer systems, second edition, Insitute for New Generation Computer Technology, Tokyo, Japan, April.

[Janas81] J.M. Janas (1981), On the feasibility of informative answers, *Advances in Data Base Theory*, vol. 1, H. Gallaire, J. Minker and J.M. Nicolas (eds.), Plenum Press, New York, 397-414.

[Jensen75] D.C. Jensen and T. Pietrzykowski (1975), Mechanizing ω-order type theory through unfication, *Theoretical Computer Science* **3**(2), 123-171.

[Kahn81] K. Kahn (1981), UNIFORM - a language based upon unification which unifies (much of) LISP, PROLOG and ACT1, *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, August 24-28, Vancouver, British Columbia, 933-939.

[Kaplan75] D. Kaplan (1975), What is Russell's theory of descriptions?, *The Logic of Grammar*, D. Davidson and G. Harman (eds.), Dickenson, Encino, California, 210-217.

[Kaplan79] S.J. Kaplan (1979), Cooperative responses from a portable natural language data base query system, HPP-79-19, Heuristic Programming Project, Stanford University, Stanford, California.

[Kaplan82] S.J. Kaplan (1982), Cooperative responses from a portable natural language query system, *Artificial Intelligence* **19**(2), 165-187.

[Katz77] J.J. Katz (1977), A proper theory of names, *Philosophical Studies* **31**, 1-80.

[Kellogg78] C. Kellogg, P. Klahr, and L. Travis (1978), Deductive planning and pathfinding for relational data bases, *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 179-200.

[Kent79] W. Kent (1979), Limitations of record-based information models, *ACM Transactions on Database Systems* **4**(1), 107-131.

[Kleer77] J. de Kleer, J. Doyle, G.L. Steele, and G.J. Sussman (1977), AMORD explicit control of reasoning, *ACM SIGART Newsletter* **64**, 116-125 [also published as *ACM SIGPLAN Notices* **12**(8)].

[Komorowski81] H.J. Komorowski (1981), Embedding PROLOG in LISP: an example of a LISP craft technique, LITH-MATH-R-1981-2, Informatics Laboratory, Linkoping University, Linkoping, Sweden, March.

[Kornfeld83] W.A. Kornfeld (1983), Equality for Prolog, *Proceedings of IJCAI-83*, August 8-12, Karlsruhe, Germany, 514-519.

[Kowalski71] R.A. Kowalski and D. Keuhner (1971), Linear resolution with selection function, *Artificial Intelligence* **2**(3&4), 227-260.

[Kowalski78] R.A. Kowalski (1978), Logic for data description, *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 77-103.

[Kowalski79] R.A. Kowalski (1979), *Logic for Problem Solving*, Artificial Intelligence Series 7, Elsevier North Holland, New York.

[Kowalski81] R.A. Kowalski (1981), Logic as a data base language, Department of Computing, Imperial College, London, England, July.

[Kowalski82] R.A. Kowalski (1982), Logic programming for the fifth generation, *Proceedings of the Fifth Generation: the dawn of the second computer age*, July 7-9, SPL International, London, England.

[LaFue82] G.M.E. LaFue (1982), Semantic integrity management of databases: a survey, LCSR-TR-32, Laboratory for Computer Science Research, Rutgers University, New Brunswick, New Jersey, October.

[Lehnert78] W. Lehnert and Y. Wilks (1978), A critical perspective on KRL, *Cognitive Science* **3**(1), 1-28.

[Leisenring69] A.C. Leisenring (1969), *Mathematical Logic and Hilbert's E-symbol*, MacDonald Technical & Scientific, London, England.

[Levien67] R.E. Levien and M.E. Maron (1967), A computer system for inference execution and data retrieval, *ACM Communications* **10**(11), 715-721.

[Lipski79] W. Lipski (1979), On semantic issues connected with incomplete information data bases, *ACM Transactions on Database Systems* **4**(3), 262-296.

[Lloyd82] J.W. Lloyd (1982), Foundations of logic programming, Technical Report 82/7, Department of Computer Science, University of Melbourne, Melbourne, Australia, August.

[Loveland78] D.W. Loveland (1978), *Automated theorem proving: a logical basis*, North-Holland, Amsterdam, The Netherlands.

[Mays81] E. Mays, S. Lanka , A. Joshi, and B.L. Webber (1981), Natural language interaction with dynamic knowledge bases: monitoring as response, *Proceedings of the Seventh International Conference on Artificial Intelligence*, August 24-28, The University of British Columbia, Vancouver, British Columbia, 61-63.

[McCarthy68] J. McCarthy (1968), Programs with common sense, *Semantic Information Processing*, M. Minsky (ed.), MIT Press, Cambridge, Massachusetts, 403-418.

[McCarthy69] J. McCarthy and P.J. Hayes (1969), Some philosophical problems from the standpoint of Artificial Intelligence, *Machine Intelligence*, vol. 4, B. Meltzer and D. Michie (eds.), American Elsevier, New York, 463-502 [Q335 M27].

[McCarthy77] J. McCarthy (1977), Epistemological problems of Artificial Intelligence, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, August 22-25, MIT, Cambridge, Massachusetts, 1083-1044.

[McDermott78a] D.V. McDermott (1978), The last survey of representation of knowledge, *Proceedings of the Artificial Intelligence and Simulation of Behaviour/Gesellschaft fur Informatik Conference on Artificial Intelligence*, Hamburg, Germany, 206-221.

[McDermott78b] D.V. McDermott (1978), Tarskian semantics, or no notation without denotation, *Cognitive Science* **2**(3), 277-282.

[McDermott80] D.V. McDermott (1980), The PROLOG Phenomenon, *ACM SIGART Newsletter* **72**, 16-20.

[McLeod81] D. McLeod (1981), Tutorial on database research, *ACM SIGMOD Record* **11**(2), 26-28.

[Melle81] W. van Melle, A.C. Scott, J.S. Bennett, and M. Peairs (1981), The EMYCIN Manual, STAN-CS-81-885, Computer Science Department, Stanford University, Stanford, California, October.

[Mendelson64] E. Mendelson (1964), *Introduction to Mathematical Logic*, Van Nostrand Reinhold, New York.

[Mercer84] R.E. Mercer and R.S. Roseberg (1984), Generating corrective answers by computing presuppositons of answers, not of questions, *Proceedings of the Fifth National Conference of the Canadian Society for the Computational Studies of Intelligence*, May 15-17, University of Western Ontario, London, Ontario, 16-19.

[Minker75] J. Minker (1975), Performing inferences over relation data bases, *Proceedings of the ACM SIGMOD International Conference on the Mangement of Data*, May, San Jose, California, W.F. King (ed.), 79-91.

[Minker78] J. Minker (1978), An experimental relational data base system based on logic, *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 107-147.

[Minker83] J. Minker (1983), On theories of definite and indefinite databases, Department of Computer Science, University of Maryland, College Park, Maryland, 53 pages.

[Minsky75] M. Minsky (1975), A framework for representing knowledge, *The Psychology of Computer Vision*, P.H. Winston (ed.), McGraw-Hill, New York, 211-277.

[Montague74a] R. Montague (1974), On the nature of certain philosophical entities, *Formal Philosophy*, R.H. Thomason (ed.), Yale University Press, 148-187 [reprinted from *The Monist* **53**(1960), 159-194].

[Montague74b] R. Montague (1974), Pragmatics and intensional logic, *Formal Philosophy*, R.H. Thomason (ed.), Yale University Press, 119-147 [reprinted from *Synthese* **22**(1970), 68-94].

[Moore74] J. Moore and A. Newell (1974), How can Merlin understand?, *Knowledge and Cognition*, L.W. Gregg (ed.), Lawrence Erlbaum Associates, Potomac, Marland, 201-252.

[Moore76] R.C. Moore (1976), D-SCRIPT, a computational theory of descriptions, *IEEE Transactions on Computers* **C-25**(4), 366-373.

[Morgan75] C.G. Morgan (1975), Automated hypothesis generation using extended inductive resolution, *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, September 3-8, Tblisi, USSR, 351-356.

[Morris69] J.B. Morris (1969), E-resolution: extension of resolution to include the equality relation, *Proceedings of the Internationl Joint Conference on Artificial Intelligence*, May 7-9, Washington, D.C., 287-294.

[Moss80] C.D.S. Moss (1980), The comparison of several PROLOG systems, *Proceedings of the Logic Programming Workshop*, July 14-16, Debrecen, Hungary, 198-200.

[Moto-oka83] T. Moto-oka (1983), Overview to the fifth generation computer system project, *ACM SIGARCH Newsletter* **11**(3), 417-422.

[Mylopoulos80] J. Mylopoulos and H.K.T. Wong (1980), Some features of the TAXIS data model, *Proceedings of the Sixth International Conference on Very Large Data Bases*, September 30-October 3, Montreal, Quebec, 399-410.

[Naish83a] L. Naish (1983), MU-PROLOG 3.0 reference manual, Department of Computer Science, University of Melbourne, Melbourne, Australia, July.

[Naish83b] L. Naish (1983), An Introduction to MU-PROLOG, Technical Report 82/2 , Department of Computer Science, Melbourne University, Melbourne, Australia, July.

[Nicolas78a] J.M. Nicolas and H. Gallaire (1978), Database: theory vs. interpretation, *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 33-54.

[Nicolas78b] J.M. Nicolas and K. Yazdanian (1978), Integrity checking in deductive data bases, *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 325-344.

[Norman79] D.A. Norman and D.G. Bobrow (1979), Descriptions: an intermediate stage in memory retrieval, *Cognitive Psychology* **11**(1), 107-123.

[Ortony77] A. Ortony and R.C. Anderson (1977), Definite descriptions and semantic memory, *Cognitive Science* **1**(1), 74-83.

[Pereira83] F.C.N. Pereira, D.H.D. Warren, L. Byrd, and L. Pereira (1983), CProlog User's Manual Version 1.2, SRI International, Menlo Park, California, 28 pages.

[Pietrzykowski73] T. Pietrzykowski (1973), A complete mechanization of second-order type theory, *ACM Journal* **20**(2), 333-365.

[Pirotte78] A. Pirotte (1978), High level data base query languages, *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 409-436.

[Prawitz60] D. Prawitz, H. Prawitz, and N. Vogera (1960), A mechanical proof procedure and it realization in an electronic computer, *ACM Journal* **7**(1&2), 102-128.

[Quine69] W.V.O. Quine (1969), *Set Theory and its Logic*, revised edition, Harvard University Press, Cambridge, Massachusetts.

[Quine80] W.V.O. Quine (1980), *From a Logical Point of View*, second edition, revised, Harvard University Press, Cambridge, Massachusetts.

[Reiter71] R. Reiter (1971), Two results on ordering for resolution with merging and linear format, *ACM Journal* **18**(4), 630-646.

[Reiter75] R. Reiter (1975), Formal reasoning and language understanding systems, *Proceedings of the First Conference on Theoretical Issues in Natural Language Processing*, June 10-13, MIT, Cambridge, Massachusetts, 175-179.

[Reiter76] R. Reiter (1976), A semantically guided deductive system for automatic theorem proving, *IEEE Transactions on Computers* **C-25**(4), 328-334.

[Reiter78a] R. Reiter (1978), Deductive question-answering on relational data bases, *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 149-177.

[Reiter78b] R. Reiter (1978), On closed world data bases, *Logic and Data Bases*, H. Gallaire and J.

Minker (eds.), Plenum Press, New York, 55-76.

[Reiter80] R. Reiter (1980), A logic for default reasoning, *Artificial Intelligence* **13**(1&2), 81-132.

[Reiter81] R. Reiter (1981), On the integrity of typed first order data bases, *Advances in Data Base Theory*, vol. 1, H. Gallaire, J. Minker and J.M. Nicolas (eds.), Plenum Press, New York, 137-157.

[Reiter83] R. Reiter (1983), Towards a logical reconstruction of relational data base theory, *On Conceptual Modelling*, M. Brodie, J. Mylopoulos and J. Schmidt (ed.), Springer-Verlag [in press].

[Roberts77] G. Roberts (1977), An implementation of PROLOG, M.Math thesis dissertation, Computer Science Department, University of Waterloo, April.

[Robinson80] A.E. Robinson and D.E. Wilkins (1980), Representing knowledge in an interactive planner, *Proceedings of the First Annual National Conference on Artificial Intelligence*, August 18-24, Stanford, California, 148-150.

[Robinson65] J.A. Robinson (1965), A machine-oriented logic based on the resolution principle, *ACM Journal* **12**(1), 23-41.

[Robinson79] J.A. Robinson (1979), *Logic: Form and Function*, Artificial Intelligence Series 6, Elsevier North Holland, New York.

[Robinson82a] J.A. Robinson and E.E. Sibert (1982), LOGLISP: motivation, design and implementaton, *Logic Programming*, A.P.I.C. Studies in Data Processing 16, K.L. Clark and S.-A. Tarnlund (eds.), Academic Press, New York, 299-313.

[Robinson82b] J.A. Robinson and E.E. Sibert (1982), LOGLISP: an alternative to PROLOG, *Machine Intelligence*, vol. 10, J.E. Hayes, D. Michie, and Y-H Pao (eds.), Ellis-Horwood, 399-419.

[Rogers71] R. Rogers (1971), *Mathematical Logic and Formalized Theories*, North Holland, New York.

[Rosenschein78] S.J. Rosenschein (1978), The production system: architecture and abstraction, *Pattern-Directed Inference Systems*, D.A. Waterman and F. Hayes-Roth (eds.), Academic Press, New York, 525-538.

[Rosser68] J.B. Rosser (1968), *Logic for mathematicians*, Chelsea, New York.

[Rosser78] J.B. Rosser (1978), *Logic for Mathematicians*, McGraw-Hill, New York.

[Rulifson72] J.F. Rulifson, J.A. Derksen, and R.J. Waldinger (1972), QA4: a procedural calculus for intuitive reasoning, Technical Note 73, Stanford Research Institute, Menlo Park, California, November.

[Schubert76] L.K. Schubert (1976), Extending the expressive power of semantic networks, *Artificial Intelligence* **7**(2), 163-198.

[Schubert79] L.K. Schubert, R.G. Goebel, and N.J. Cercone (1979), The structure and organization of a semantic net for comprehension and inference, *Associative Networks: Representation and use of Knowledge by Computers*, N.V. Findler (ed.), Academic Press, New York, 121-175.

[Shapiro79] S.C. Shapiro (1979), Numerical quantifiers and their use in reasoning with negative information, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, August 20-23, Tokyo, Japan, 791-796.

[Steels80] L. Steels (1980), Description types in the XPRT-system, *Proceedings of the AISB-80 Conference on Artificial Intelligence*, July 1-4, Amsterdam, Holland, (STEELS 1-9).

[Stickel75] M.E. Stickel (1975), A complete unification algorithm for associative-commutative functions, *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, September 3-8, Tblisi, USSR, 71-76.

[Stonebraker75] M. Stonebraker (1975), Implementation of integrity constraints by query modification, *Proceedings ACM SIGMOD International Conference on Management of Data*, May, San Jose, California, 65-78.

[Stonebraker76] M. Stonebraker, E. Wong, P. Kreps, and G. Held (1976), The design and implementation of INGRES, *ACM Transactions on Database Systems* **1**(3), 189-222.

[Stonebraker80] M. Stonebraker (1980), Retrospective on a database system, *ACM Transactions on Database Management* **5**(2), 225-240.

[Stoy77] J.E. Stoy (1977), *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Massachusetts.

[Sussman71] G.J. Sussman, T. Winograd, and E. Charniak (1971), Micro-planner reference manual, AI Memo 203A, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December.

[Sussman72] G.J. Sussman and D.V. McDermott (1972), Why conniving is better than planning, AI Memo 255A, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, April.

[Thompson69] F.B. Thompson, P.C. Lockemann, B. Dostert, and R.S. Deverill (1969), REL:a rapidly extensible language system, *Proceedings of the Twenthy-Fourth ACM National Conference*, August 26-28, San Francisco, California, 399-417.

[Tsichritzis77] D.C. Tsichritzis and F.H. Lochovsky (1977), *Data Base Management Systems*, Academic Press, New York.

[Turner82] D.A. Turner (1982), Recursion equations as a programming language, *Functional programming and its applications: an advanced course*, J. Darlington, P. Henderson, and D.A. Turner (eds.), Cambridge University Press, Cambridge, England, 1-28.

[Vassiliou79] Y. Vassiliou (1979), Null values in database management: a denotational semantics approach, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May, Boston, Massachusetts, 162-169.

[Wang52] H. Wang (1952), Logic of many-sorted theories, *Journal of Symbolic Logic* **17**(2), 105-116.

[Warren82] D.H.D Warren (1982), A view of the fifth generation and its impact, *AI Magazine* **3**(4), 34-39.

[Weyrauch80] R.W. Weyrauch (1980), Prolegomena to a theory of mechanized formal reasoning, *Artificial Intelligence* **13**(1&2), 133-170.

[Williams83] M.H. Williams, J.C. Neves, and S.O. Anderson (1983), Security and integrity in logic data bases using query-by-example, *Proceedings of Logic Programming Workshop '83*, June 26 - July 1, Algarve, Portugal, 304-340.

[Wong77] H.K.T. Wong and J. Mylopoulos (1977), Two views of data semantics: a survey of data models in Artificial Intelligence and Database Management, *INFOR* **15**(3), 344-383.

[Woodfill81] J. Woodfill, P. Siegal, J. Ranstrom, M. Meyer, and E. Allman (1981), INGRES version 7 reference manual, Electronics Research Laboratory, University of California, Berkeley, California, August.

Appendix 1
DLOG implementation in Waterloo Prolog

Below are the source code listings for the current Waterloo Prolog implementation of DLOG system.

```
/*USAGE

 RELATIONS:  all
          _all
          _allset

 DESCRIPTION: computes all bindings of freevar in the current
      PROLOG/CMS data base. _all produces a list in the
      form 'a.b.c. ... .NIL', all produces a list of the
      form 'a & b & c &... z.'

 NEEDS: element    SETFNS1
      _elementset
      _sort       SORT1

 USED IN: extension  EXT1

*/

 all( *expr, *list ) <-
    ADDAX( _all( NIL ) )
   & ~gen( *expr )
   & DELAX( _all( *res ) )
   & remNIL( *res, *list ).

/*
*/

 gen( lambda( *el, *expr) ) <-
    *expr
   & _all( *sol )
   & ~element( *el, set( *sol ) )
   & DELAX( _all( *sol ) )
   & ADDAX( _all( *el & *sol ) )
   & FAIL.

/*
```

```
*/

 remNIL( *x & NIL, *x ).

 remNIL( *x & *y, *x & *z ) <-
    remNIL( *y, *z ).

/*
*/

 _all( lambda( *x, *expr ), *list ) <-
    ADDAX( __all( NIL ) )
   & ~_gen( lambda( *x, *expr ) )
   & DELAX( __all( *list ) ).

/*
*/

 _gen( lambda( *x, *expr ) ) <-
    *expr
   & __all( *sol )
   & ~_element( *x, *sol )
   & DELAX( __all( *sol ) )
   & ADDAX( __all( *x.*sol ) )
   & FAIL.

/*
*/

 _allset( lambda( *x, *expr ), *list ) <-
    ADDAX( __allset( NIL ) )
   & ~_genset( lambda( *x, *expr ) )
   & DELAX( __allset( *list ) ).

/*
*/

 _genset( lambda( *x, *expr ) ) <-
    *expr
   & __allset( *sol )
   & _sort( *x, *xs )
   & ~_elementset( *xs, *sol )
   & DELAX( __allset( *sol ) )
   & ADDAX( __allset( *xs.*sol ) )
   & FAIL.
```

```
/*USAGE

 RELATION: assert

 DESCRIPTION: assert applies integrity constraints, then adds the
      clause to the global data base.

 NEEDS: verify_ic     IC1
       print         UTILS1
       atomic

 USED IN: input        INPUT1
        transaction TRANS1

*/

assert( *a <= *c ) <-      /* integrity constraint */
   verify_ic( *a <= *c )
  & print( 'Assert constraint: '.*a.'<='.*c )
  & ADDAX( *a <= *c ).

assert( *c <- *a ) <-      /* implication */
   verify_ic( *c <- *a )
  & print( 'Assert implication: '.*c.'<-'.*a )
  & ADDAX( *c <- *a ).

assert( *a ) <-      /* atom */
   atomic( *a )
  & verify_ic( *a )
  & print( 'Assert atom: '.*a )
  & ADDAX( *a ).

assert( *a ) <-
   print( 'Constraints failed for '.*a ).
```

```
/*USAGE

 RELATION: browse
        browse_predicate
        browse_skeleton
        browse_topic

 DESCRIPTION: User feature to browse through the DLOG data base
     contents.

 NEEDS: for    UTILS1
      user
      update
      print
      printl
      prompt
      read_string
      menu_position
      _all     ALL1

 USED IN: parse     PARSE1

*/

browse <-
   menu_position( 'Topics'
             .'Constraints'
             .'User predicates'
             .'System predicates'
             .'Enter predicate'
             .'Enter skeleton'
             .NIL, *n )
 & browse_1( *n )
 & user( 'Continue browsing?' )
 & RETRY( browse ).

browse.

/*
*/

browse_1( 1 ) <-
   browse_topic.

browse_1( 2 ) <-
   get_skeleton( *sk )
 & browse_skeleton( *sk<=* ).

browse_1( 3 ) <-
   browse_predicate( user_predicate, 2 ).

browse_1( 4 ) <-
   browse_predicate( system_predicate, 2 ).
 browse_1( 5 ) <-
    get_predicate( *p, *n )
  & browse_predicate( *p, *n ).

 browse_1( 6 ) <-
    get_skeleton( *sk )
  & browse_skeleton( *sk ).

/*
*/

 get_predicate( *p, *n ) <-
    prompt( 'Predicate? ' )
  & read_string( *p )
  & verify_predicate( *p, *n )
  & /.

 get_predicate( *, * ) <-
    RETRY( browse ).

/*
*/

 get_skeleton( *sk ) <-
    prompt( 'Skeleton (end with .)? ' )
  & read_skeleton( *sk )
  & verify_skeleton( *sk )
  & /.

 get_skeleton( * ) <-
    RETRY( browse ).

/*
*/

 get_topic( *t ) <-
    prompt( 'Topic? ' )
  & read_string( *i )
  & verify_topic( *i, *t )
  & /.

 get_topic( * ) <-
    user( 'Browse topics?' )
  & browse_predicate( topic, 1 ).

 get_topic( * ) <-
    RETRY( browse ).

/*
*/

 verify_predicate( *p, *n ) <-
```

```
    user_predicate( *p, *n )
  | system_predicate( *p, *n ).

verify_predicate( *p, *n ) <-
    print( 'Unknown predicate: '.*p )
  & user( 'Retry?' )
  & RETRY( get_predicate( *, * ) ).

/*
*/

browse_predicate( *p, *n ) <-
    ADDAX( _bi( 1 ) )
  & browse_predicate_1( *p, *n )
  & DELAX( _bi( * ) ).

browse_predicate_1( *p, *n ) <-
    _bi( *i )
  & SUM( *i, 3, *ne )
  & for( *i, *ne, 1, _for( *x.* )
                 & AXN( *p, *n, *ax, *x )
                 & print( *ax ) )
  & SUM( *ne, 1, *nx )
  & AXN( *p, *n, *, *nx )
  & user( 'More?' )
  & update( _bi( *i ), _bi( *nx ) )
  & RETRY( browse_predicate_1( *, * ) ).

browse_predicate_1( *, * ).

/*
*/

browse_skeleton( *sk ) <-
    ADDAX( _bi( 1 ) )
  & browse_skeleton_1( *sk )
  & DELAX( _bi( * ) ).

browse_skeleton_1( *sk ) <-
    AX( *sk, *sk )
  & print( *sk )
  & _bi( *i )
  & SUM( *i, 1, *n )
  & update( _bi( *i ), _bi( *n ) )
  &
  & SUM( *i, 1, *n )
  & update( _bi( *i ), _bi( *n ) )
  & GE( *n, 4 )
  & update( _bi( * ), _bi( 1 ) )
  & ~user( 'More?' ).

browse_skeleton_1( * ).
```

```
/*
*/

browse_topic <-
   get_topic( *t )
 & _all( lambda( *x, topic_category( *t, *x ) ), *l )
 & print( 'Predicates relevant to '.*t )
 & printl( *l )
 & user( 'Another topic?' )
 & RETRY( browse_topic ).

browse_topic.
```

```
/*USAGE

  RELATION: create_context
           destroy_context
           local_ctx

  DESCRIPTION: create and destroy data base context by adding some
       assertions to the current data base. Create_ctx and
       destroy generate unique context tag, which is used
       to expliciting create and destroy contexts. Local_ctx
       creates a context, and destroys it when finished.

  NEEDS: gensym      UTILS1
         copy
         deterministic

  USED IN: many places

*/

 create_ctx( *tag, *expr ) <-
    gensym( ctx, *tag )
  & deterministic( create_ctx_1( *expr ) )
  & ADDAX( _ctx( *tag, *expr ) ).

 create_ctx_1( *lit & *rest ) <-
    ADDAX( *lit )
  & create_ctx_1( *rest ).

 create_ctx_1( *lit ) <-
    ADDAX( *lit ).

/*
*/

 destroy_ctx( *tag ) <-
    ATOM( *tag )
  & _ctx( *tag, *expr )
  & deterministic( destroy_ctx_1( *expr ) )
  & DELAX( _ctx( *tag, * ) ).

 destroy_ctx_1( *lit & *rest ) <-
    DELAX( *lit )
  & destroy_ctx_1( *rest ).

 destroy_ctx_1( *lit ) <-
    DELAX( *lit ).

/*
*/

 local_ctx( *assertions, *goals ) <-
    deterministic( create_local_ctx( *assertions ) )
  & deterministic( *goals )
  & deterministic( destroy_local_ctx( *assertions ) ).

local_ctx( *assertions, * ) <-
    destroy_local_ctx( *assertions )
  & FAIL.

/*
*/

create_local_ctx( *lit & *rest ) <-
    ADDAX( *lit )
  & create_local_ctx( *rest ).

create_local_ctx( *lit ) <-
    ADDAX( *lit ).

/*
*/

destroy_local_ctx( *lit & *rest ) <-
    DELAX( *lit )
  & destroy_local_ctx( *rest ).

destroy_local_ctx( *lit ) <-
    DELAX( *lit ).
```

```
/*USAGE

 RELATION: derivable

 DESCRIPTION: Simulates PROLOG/CMS derivations. Handles goals
     with set terms.

 NEEDS: atom      SYNTAX1
      set_constant
      set_description
      debug    UTILS1
      deterministic
      unify    UNIFY1
      system_predicate  PREDS1
      prolog_predicate
      set_predicate

 USED IN: verify_ic      IC1
        implies       META1
        satisfied

*/

derivable( derivable( *g ) ) <-     /* peel off one level of simulation */
   deterministic( derivable( *g ) ).

derivable( *g & *grest ) <-
   derivable( *g )
 & derivable( *grest ).

derivable( *g | *grest ) <-
   derivable( *g )
 | derivable( *grest ).

derivable( *g ! *grest ) <-
   derivable( *g )
 ! derivable( *grest ).

/*
 individual goals
*/

derivable( ~*g ) <-
   atom( *g )
 & ~derivable_1( *g ).

derivable( *g ) <-
   atom( *g )
 & derivable_1( *g ).

/*
 derivable_1 splits 3 ways: system goals, goals requiring set expansion, and
                    DLOG goals.
```

```
*/

 derivable_1( *g ) <-
    CONS( *p.*t, *g )
  & ( set_predicate( *p ) | ~contains_set_term( *g ) )
  & system_predicate( *p, * )
  & debug( sys, *g )
  & *g.

 derivable_1( *g ) <-
    CONS( *p.*t, *g )
  & contains_set_term( *t )
  & ~prolog_predicate( *p )
  & ~set_predicate( *p )
  & debug( set, *g )
  & s_derivable( *g ).

 derivable_1( *g ) <-
    CONS( *p.*t, *g )
  & ~system_predicate( *p, * )
  & ( set_predicate( *p ) | ~contains_set_term( *t ) )
  & debug( ind, *g )
  & i_derivable( *g ).

/*
 derivation of non-set predicates requiring set expansion
*/

 s_derivable( *g ) <-
    CONS( *p.*t, *g )
  & _substitute( *x, set( *x : *e1, *e2 ), *t, *nt )
  & CONS( *p.*nt, *ng )
  & extension( set( *x : *e1 & *ng, *e2 ), * ).


 s_derivable( *g ) <-
    CONS( *p.*t, *g )
  & _substitute( *x, set( *x : *e1 ), *t, *nt )
  & CONS( *p.*nt, *ng )
  & _extension( set( *x : *e1 ), *ext )
  & map( lambda( *x, *ng ), *ext ).

/*
i_derivable does derivation of DLOG atoms. Note that if any set terms appear,
  ^?*they are treated as terms, not aggregates as in s_derivable.
*/

i_derivable( *g ) <-
   retrieve( *g, *db )
  & atom( *db )
  & unify( *g, *db ).

i_derivable( *g ) <-
```

```
    retrieve( *g, *csq <- *ant )
  & unify( *g, *csq )
  & debug( recurse, *ant )
  & derivable( *ant ).

/*
*/

retrieve( *g, *db ) <-
    AX( *g, *db )
  & debug( retrieve, *db ).

/*
*/

contains_set_term( *x.*y ) <-
    deterministic( set_constant( *x ) | set_description( *x ) )
  | contains_set_term( *y ).
```

```
/*USAGE

 RELATION: extension
         _extension

 DESCRIPTION: extension will produce the extensions of DLOG
      individual and set terms. The relation _extension
      gives terms of the form 'a.b. ... .NIL', extension
      produces terms of the form 'a & b ... & z'

 NEEDS: _union      SETFNS1
       _intersection
       _psubset
       aggregate   SYNTAX1
       individual_constant
       all        ALL1
       _all
       apply      META1
       cons_amp     UTILS1

 USED IN: unify     UNIFY1

*/
/*
 individuals and individual sets
*/

_extension( *x , *x.NIL ) <-
   individual_constant( *x ).

_extension( *ag, *ext ) <-
   aggregate( *ag )
  & _ext( *ag, *ext )

_extension( an( *v, *expr ), *v.NIL ) <-
  *expr.

_extension( the( *v, *expr ), *i.NIL ) <-
   _all( lambda( *v, *expr ), *e )
  & bind( *i.NIL, *e )
  & individual_constant( *i ).

_extension( set( *x ; set( *y ) : *expr1 & *expr2 ), *ext ) <-
   _all( lambda( *x, *expr1 ), *ext1 )
  & _psubset( *ext, *ext1 )
  & cons_amp( *ext, *ext2 )
  & apply( lambda( set( *y ), *expr2 ), *ext2 ).

_extension( set( *x : *expr ),  *ext ) <-
   _all( lambda( *x, *expr ), *ext ).

_extension( set( *ag ), *ext ) <-
   aggregate( *ag )
```

```
    & _ext( *ag, *ext ).

 /*
    aggregates
 */

 _ext( *x, * ) <-
     individual_variable( *x )
   & /( _ext( *, * ) )
   & FAIL.

 _ext( *x, *x.NIL ) <-
     individual_constant( *x ).

 _ext( *x | *y, *ex ) <-      /* ? | ? */
     _ext( *x, *ex ).

 _ext( *x | *y, *ey ) <-
     _ext( *y, *ey ).

 _ext( *x | *y, *exy ) <-
     _ext( *x, *ex )
   & _ext( *y, *ey )
   & _union( *ex, *ey, *exy ).

 _ext( *x ! *y, *ex ) <-      /* ? ! ? */
     _ext( *x, *ex ).

 _ext( *x ! *y, *ey ) <-
     _ext( *x, *ex )
   & _ext( *y, *ey )
   & _intersection( *ey, *ex, NIL ).

 _ext( *x & *y, *exy ) <-      /* ? & ? */
     _ext( *x, *ex )
   & _ext( *y, *ey )
   & _union( *ex, *ey, *exy ).

/*
*/

extension( set( *x ), set( *ext ) ) <-
    _extension( set( *x ), *list )
  & cons_amp( *list, *ext ).

extension( set( *x : *e ), set( *ext ) ) <-
    _extension( set( *x : *e ), *list )
  & cons_amp( *list, *ext ).

extension( set( *x ; set( *y ) : *e1 & *e2 ), set( *ext ) ) <-
    _extension( set( *x ; set( *y ) : *e1 & *e2 ), *list )
  & cons_amp( *list, *ext ).
```

```
extension( *x, *ext ) <-
   _extension( *x, *list )
  & cons_amp( *list, *ext ).
```

```
/*USAGE

  RELATION: verify_ic

  DESCRIPTION: applies integrity constraints to new assertions.

  NEEDS: print      UTILS1
         bind
         atomic
         derivable  DERIVE21

  USED IN: assert      ASSERT1
           assertTRANS TRANS1

*/

 verify_ic( *a <= *c ) <-
    ~AX( *a, * ).

 verify_ic( *a <= *c ) <-
    _all( lambda( *x,
               & AX( *a, *y ) & atomic( *y ) & bind( *a <= *c, *y <= *x )),
                *ics )
  & verify_ic_1( *ics ).

 verify_ic( *c <- *t ).      /* no test yet for IMPLs */

 verify_ic( *a ) <-
    ~AX( *a <= *, *a <= * ).

 verify_ic( *a ) <-
    _all( lambda( *ic, AX( *a <= *, *a <= *ic ) ), *ics )
  & verify_ic_1( *ics ).

/*
*/

 verify_ic_1( *ic.NIL ) <-     /* isolate constraint */
    verify_ic_2( *ic ).

 verify_ic_1( *ic.*icrest ) <-
    verify_ic_2( *ic )
  & verify_ic_1( *icrest ).

/*
*/

 verify_ic_2( *c1 & *crest ) <-     /* isolate conjuncts */
    verify_ic_3( *c1 )
  & verify_ic_2( *crest ).

 verify_ic_2( *c1 ) <-
    verify_ic_3( *c1 ).
```

```
/*
*/

verify_ic_3( *c ) <-       /* test constraint */
   derivable( *c ).

verify_ic_3( *c ) <-       /* constraint fails */
      print( 'Failed constraint: '.*c )
    & /( verify_ic( * ) )
    & FAIL.
```

```
/*USAGE

 RELATION: input

 DESCRIPTION: Interprets DLOG system commands, applies read macros,
     and forwards transformed inputs to the other processors.

 NEEDS: assert     ASSERT1
      browse     BROWSE1
      transaction TRANS1
      query      QUERY1
      parse      PARSE1
      deterministic META1
      print      UTILS1
      load

 USED IN: start     START1

*/

input( version ) <-
   print( 'DLOG 1.0' ).

input( stop ) <-
   print( 'EXIT DLOG' )
  & /( restart )
  & FAIL.

input( load ) <-
   deterministic( load ).

input( browse ) <-
   deterministic( browse ).

input( transaction ) <-
   deterministic( transaction ).

input( *c ) <-
   user_command( *c )
  & deterministic( *c ).

input( <- *q ) <-
   deterministic( query( *q ) ).

input( *a ) <-
   deterministic( parse( *a, *at ) )
  & deterministic( assert( *at ) ).

input( *i ) <-
   print( 'Input ignored: '.*i ).
```

```
/*USAGE

 RELATIONS: eqlambda
           eqset
           eqaggregate
           eqext
           implies
           extends
           satisfied
           apply
           map

 DESCRIPTION: implements meta logical operations on DLOG terms.

 NEEDS: _remove    SETFNS1
       create_ctx    CTX1
       destroy_ctx
       local_ctx
       sort    SORT1
       _allset    ALL1
       derivable    DERIVE1
       aggregate    SYNTAX1
       unify_1    UNIFY1

 USED IN: unify    UNIFY1
         verify_ic    IC1

*/

eqlambda( lambda( *v1, *l1) , lambda( *v2, *l2 ) ) <-
   implies( *l1, *l2 )
  & implies( *l2, *l1 ).

/*
*/

eqset( set( *s ), set( *s ) ).

eqset( set( *x : *expr1 ), set( *y : *expr2 ) ) <-
   eqlambda( lambda( *x, *expr1 ), lambda( *y, *expr2 ) ).

eqset( set( *x ; set( *y ) : *expr1 & *expr2 ), set( *z ; set( *w ) : *expr3, *expr4 ) ) <-
   eqlambda( lambda( *x, *expr1 ), lambda( *z, *expr3 ) )
  & eqlambda( lambda( set( *y ) , *expr2 ), lambda( set( *w ), *expr4 ) ).

eqset( set( *s1 ), set( *s2 ) ) <-
   aggregate( *s1 )
  & aggregate( *s2 )
  & eqaggregate( *s1, *s2 ).

eqset( set( *s1 ), set( *s2 ) ) <-
   sort( *s1, *s )
  & sort( *s2, *s ).
```

```
/*
*/

 eqaggregate( *ag1, *ag2 ) <-
     _allset( lambda( *e1, _extension( *ag1, *e1 ) ), *ext1 )
   & _allset( lambda( *e2, _extension( *ag2, *e2 ) ), *ext2 )
   & eqext( *ext1, *ext2 ).

/*
*/

 eqext( *s, *s ).

 eqext( *x.*r1, *x.*r2 ) <-
     eqext( *r1, *r2 ).

 eqext( *x.*r1, *y.*r2 ) <-
     _remove( *x, *r2, *rx2 )
   & _remove( *y, *r1, *ry1 )
   & eqext( *rx2, *ry1 ).

/*
*/

 implies( *dj | *rest, *csq ) <-
     deterministic( implies( *dj, *csq )
                  & implies( *rest, *csq ) ).

 implies( *xj ! *rest, *csq ) <-
     deterministic( implies( *xj, *csq )
                  & implies( *rest, *csq ) ).

 implies( *cj & *rest, *csq ) <-
     local_ctx( *cj, implies( *rest, *csq ) ).

 implies( *ant, *csq ) <-
     local_ctx( *ant, derivable( *csq ) ).

/*
*/

 extends( *cj & *rest, *hy ) <-
     deterministic( extends( *cj, *hy ) | extends( *rest, *hy ) ).

 extends( *dj | *rest, *hy ) <-
     deterministic( extends( *dj, *hy ) | extends( *rest, *hy ) ).

 extends( *xj ! *rest, *hy ) <-
     deterministic( extends( *xj, *hy ) | extends( *rest, *hy ) ).

 extends( *db, *hy ) <-
     local_ctx( ( unify_1( *a.*qts, set( *x : *e1, *e2 ).*dbts ) <-
                ATOM( *a )
              & apply( lambda( *x, *e1 ), *a )
              & unify_1( *qts, *dbts ) )
          & ( unify_1( an( *y, *e3 ).*qts, set( *z : *e4, *e5 ).*dbts ) <-
                eqlambda( lambda( *y, *e3 ), lambda( *z, *e4 ) )
              & unify_1( *qts, *dbts ) ),
    CONS( *p.*hyt, *hy )
  & CONS( *p.*dbt, *db )
  & unify_1( *hyt, *dbt ) ).

/*
*/

satisfied( *x, lambda( *x, *expr ) ) <-
    derivable( *expr )
  & /.

satisfied( *x, lambda( *x, *expr ) ) <-
    print( 'Failed to satisfy: '.*expr )
  & FAIL.

/*
*/

apply( lambda( *x, *y), *z ) <-
    copy( *x.*y, *z.*w )
  & *w.

/*
*/

map( lambda( *var, *pred ), *arg.*rest ) <-
    copy( lambda( *var, *pred ), lambda( *arg, *test ) )
  & *test
  & map( lambda( *var, *pred ), *rest ).


map( lambda( *var, *pred ), NIL ).
```

```
/*USAGE

  RELATION: N/A

  DESCRIPTION: a collection of operator definitions for DLOG.

  NEEDS: N/A

  USED IN: ( must be loaded before any DLOG/CMS relations, since
     operators define DLOG/CMS syntax ).

*/

/*
  The special operators used are as follows:

     !     - infix operator used for exclusive or,
             e.g., req(CS,CS111!CS115)

     < >  - alternative way to delimit DLOG set terms,
            e.g., input parser will convert <CS115&CS111>
            to set(CS115&CS111)

     { } - alternative way to delimit DLOG set terms,
            e.g., input parser will convert {CS115&CS111}
            to set(CS115&CS111)

     :     - variable delimiter in set specification,
             e.g., { *x : p( *x ) }

     ;     - variable delimiter in set specification,
             e.g., { *x ; set( *y ) : p( *x ) & q( set( *y ) ) }

     ?     - prefix operator for queries

     <=     - infix operator for integrity constraints


*/

OP( '<=' RL, 10 ).      /* non-Horn constraint */

OP( !, LR, 20 ).     /* exclusive or operator */

*x ! *y <- *x & ~*y.      /* semantics of ! */
*x ! *y <- ~*x & *y.

OP( :, LR, 15 ).     /* set syntax delimiter */
OP( ;, LR, 15 ).     /* set syntax delimiter */
OP( '{', PREFIX, 11 ).     /* set delimiters */
OP( '}', SUFFIX, 11 ).
```

```
/*USAGE

 RELATION: parse

 DESCRIPTION: Verifies the syntax of input assertions, applying various
      input transformations specified as input constraints.

 NEEDS: local_ctx     CTX1
        print         UTILS1
        extension     EXT1
        _substitute   SETFNS1

 USED IN: input     INPUT1

*/

parse( *i, *o ) <-
   local_ctx( input_mode
         & _input( *i, * ),
           update( _input( *i, * ), _input( *i, *i ) )
         & assertion( *i )
         & _input( *, *o ) ).

/*
  input constraints ( used for assertion time inference )
*/


input_constraint( the( *x, *e ) ) <-
   extension( the( *x, *e ), *c )
  & user( 'Is '.*c.' the intended referent of '.the( *x, *e ).'?' )
  & _input( *i, * )
  & CONS( *p.*l, *i )
  & _substitute( *c, the( *x, *e ), *l, *nl )
  & CONS( *p.*nl, *n )
  & update( _input( *i, * ), _input( *i, *n ) )
  & /.

input_constraint( the( *x, *e ) ) <-
   print( 'No known referent for '.the( *x, *e ) )
  & /
  & FAIL.
```

```
/*USAGE

 RELATIONS: system_predicate
         set_predicate

 DESCRIPTION: a list of DLOG predicates which may appear in
      DLOG user data bases. Equivalent to the primitives
      of DLOG's data manipulation/definition language.

      Currently uppercase predicates are the PROLOG/CMS
      predicates accessible at the DLOG user interface.

 NEEDS:

 USED IN: assertion  SYNTAX1
        derivable    DERIVE1
        browse     BROWSE1

*/

system_predicate( union, 3 ).
system_predicate( intersection, 3 ).
system_predicate( subset, 2 ).
system_predicate( element, 2 ).
system_predicate( cardinality, 2 ).
system_predicate( difference, 3 ).
system_predicate( all, 2 ).
system_predicate( extension, 2 ).
system_predicate( extends, 2 ).
system_predicate( local_ctx, 2 ).
system_predicate( satisfied, 2 ).
system_predicate( digit_suffix, 2 ).
system_predicate( nth_digit, 3 ).
system_predicate( letter_prefix, 2 ).
system_predicate( nth_char, 3 ).
system_predicate( concatenate, 3 ).
system_predicate( length, 2 ).
system_predicate( apply, 2 ).
system_predicate( user_select, 2 ).
system_predicate( load, 0 ).

system_predicate( user_predicate, 2 ).
system_predicate( user_command, 1 ).
system_predicate( topic, 1 ).
system_predicate( subtopic, 2 ).
system_predicate( topic_category, 2 ).
system_predicate( topic_equivalent, 2 ).
system_predicate( assertion, 1 ).
system_predicate( implication, 1 ).
system_predicate( constraint, 1 ).
system_predicate( clause, 1 ).
system_predicate( literal, 1 ).
system_predicate( atom, 1 ).
system_predicate( term, 1 ).
system_predicate( lambda, 1 ).
system_predicate( individual, 1 ).
system_predicate( individual_description, 1 ).
system_predicate( individual_constant, 1 ).
system_predicate( individual_variable, 1 ).
system_predicate( individual_aggregate, 1 ).
system_predicate( set, 1 ).
system_predicate( set_description, 1 ).
system_predicate( set_constant, 1 ).
system_predicate( set_variable, 1 ).
system_predicate( definite_individual, 1 ).
system_predicate( indefinite_individual, 1 ).
system_predicate( definite_set, 1 ).

system_predicate( SUM, 3 ).
system_predicate( DIFF, 3 ).
system_predicate( QUOT, 3 ).
system_predicate( PROD, 3 ).
system_predicate( GE, 2 ).
system_predicate( LE, 2 ).
system_predicate( LT, 2 ).
system_predicate( GT, 2 ).
system_predicate( EQ, 2 ).
system_predicate( ADDAX, 1 ).
system_predicate( DELAX, 1 ).
system_predicate( INT, 1 ).
system_predicate( ATOM, 1 ).

/*
*/

set_predicate( union ).
set_predicate( intersection ).
set_predicate( subset ).
set_predicate( element ).
set_predicate( difference ).
set_predicate( cardinality ).
set_predicate( all ).
set_predicate( extension ).
```

```
/*USAGE

 RELATION: query

 DESCRIPTION: processes DLOG queries, as prepared by parse.

 NEEDS: derivable    DERIVE1
        print     UTILS1
        extension  EXT1

 USED IN: parse  PARSE1

*/

query( set( *x : *e ) ) <-
   print( 'Query: '.set( *x : *e ) )
 & extension( set( *x : *e ), *ext )
 & print( 'Result: '.*ext ).

query( set( *x ; set( *y ) : *e1 & *e2 ) ) <-
   print( 'Query: '.set( *x ; set( *y ) : *e1 &  *e2 ) )
 & extension( set( *x ; set( *y ) : *e1 &  *e2 ), *ext )
 & print( 'Result: '.*ext ).

query( *t ) <-
   ~set( *t )
 & print( 'Query: '.*t )
 & local_ctx( query_mode, derivable( *t ) )
 & print( 'Succeeds: '.*t ).

query( *t ) <-
   print( 'Not deducible' ).
```

```
/*USAGE

 RELATIONS: element       _element
                        _elementset
                        _seteq
          cardinality   _cardinality
          subset        _subset
                        _psubset
          intersection  _intersection
          union         _union
          difference    _difference
          remove        _remove
                        _tail
                        _substitute

 DESCRIPTION: user and system set relations. User relations
      operate on sets represented as conjuncts, viz.
      A & B & ...& Z. System relations use PROLOG/CMS
      lists for sets, viz. A.B.C.NIL.

 NEEDS: _sort   SORT1
       bind   UTILS1

 USED IN: many places

*/

 _element( *x, *x.*y ).

 _element( *x, *y.*z ) <-
    _element( *x, *z ).

/*
*/

 _elementset( *x, *y.*z ) <-
    _seteq( *x, *y ).

 _elementset( *x, *y.*z ) <-
    _elementset( *x, *z ).

/*
*/

 _seteq( *x, *x ).

 _seteq( *x, *y ) <-
    _sort( *x, *xs )
  & _sort( *y, *xs ).

/*
*/
```

```
 _cardinality( NIL, 0 ).

 _cardinality( *x.*y, *n ) <-
     _cardinality( *y, *n1 )
   & SUM( *n1, 1, *n ).

/*
*/

 _subset( NIL, * ).

 _subset( *x.*y, *z ) <-
     _remove( *x, *z, *w )
   & _intersection( *y, *z, *w ).

/*
*/

 _psubset( NIL, * ).

 _psubset( *x.*y, *z ) <-
    _tail( *x, *z, *w )
  & _psubset( *y, *w ).

/*
*/

 _intersection( *x, *x, *x ).

 _intersection( NIL, *x, NIL ).

 _intersection( *x, NIL, NIL ).

 _intersection( *x.*y, *z, *w ) <-
     _remove( *x, *z, * )
   & _intersection( *y, *z, *w ).

 _intersection( *x.*y, *z, *x.*u ) <-
     _remove( *x, *z, *w )
   & _intersection( *y, *w, *u ).

/*
*/

 _union( *x, *x, *x ).

 _union( NIL, *x, *x ).

 _union( *x, NIL, *x ).

 _union( *x.*y, *z, *x.*v ) <-
     _remove( *x, *z, * )
   & _union( *y, *z, *v ).
 _union( *x.*y, *z, *v) <-
     _remove( *x, *z, *w )
   & _union( *y, *w, *v ).

/*
*/

 _difference( NIL, *x, NIL ).

 _difference( *x, NIL, *x ).

 _difference( *x, *y, *x ) <-
     _intersection( *x, *y, NIL ).

 _difference( *x.*y, *z, *x.*v ) <-
     ¬_remove( *x, *z, * )
   & _difference( *y, *z, *v ).

 _difference( *x.*y, *z, *v ) <-
     _remove( *x, *z, *w )
   & _difference( *y, *w, *v ).

/*
*/

 _remove( *x, *x.*y, *y ).

 _remove( *x, *y.*rest, *y.*z ) <-
     _remove( *x, *rest, *z ).

/*
*/

 _tail( *x, *x.*y, *y ).

 _tail( *x, *y.*z, *w ) <-
     _tail( *x, *z, *w ).

/*
*/

 _substitute( *x, *y, *z.*w, *x.*w ) <-
     ¬VAR( *z )
   & bind( *y, *z ).

 _substitute( *x, *y, *z.*w, *z.*u ) <-
     _substitute( *x, *y, *w, *u ).

/*
*/

element( *x, set( *x ) ) <-
```

```
    ATOM( *x ).

 element( *x, set( *x & *y ) ) <-
    ATOM( *x ).

 element( *x, set( *y & *z ) ) <-
    element( *x, set( *z ) ).

/*
*/

 cardinality( set( NIL ), 0 ).

 cardinality( set( *x ), 1 ) <-
    ~bind( *x, NIL )
  & ATOM( *x ).

 cardinality( set( *x & *y ), *n ) <-
    cardinality( set( *y ), *n1 )
  & SUM( *n1, 1, *n ).

/*
*/

 subset( set( NIL ), set( * ) ).

 subset( set( *x ), set( *x ) ).

 subset( set( *x ), set( *x & *y ) ).

 subset( set( *x ), set( *y & *z ) ) <-
    subset( set( *x ), set( *z ) ).

 subset( set( *x & *y ), set( *x & *z ) ) <-
    subset( set( *y ), set( *z ) ).

 subset( set( *x & *y ), set( *z & *w ) ) <-
    ~bind( *x, *z )
  & element( *x, set( *w ) )
  & subset( set( *y ), set( *z & *w ) ).

/*
*/

 intersection( set( *x ), set( *x ), set( *x ) ).

 intersection( set( NIL ), set( *x ), set( NIL ) ).

 intersection( set( *x ), set( NIL ), set( NIL ) ).

 intersection( set( *x ), set( *z ), set( *x ) ) <-
    ATOM( *x )
  & element( *x, set( *z ) ).
```

```
intersection( set( *x ), set( *z ), set( NIL ) ) <-
    ATOM( *x )
  & ~element( *x, set( *z ) ).

intersection( set( *x & *y ), set( *x & *z ), set( *x & *w ) ) <-
    intersection( set( *y ), set( *z ), set( *w ) ).

intersection( set( *x & *y ), set( *z & *w ), set( *u ) ) <-
    ~EQ( *x, *z )
  & ~element( *x, set( *w ) )
  & intersection( set( *y ), set( *z & *w ), set( *u ) ).

intersection( set( *x & *y ), set( *z & *w ), set( *x & *u ) ) <-
    ~EQ( *x, *z )
  & element( *x, set( *w ) )
  & intersection( set( *y ), set( *z & *w ), set( *u ) ).

/*
*/

union( set( *x ), set( *x ), set( *x ) ).

union( set( NIL ), set( *x ), set( *x ) ).

union( set( *x ), set( NIL ), set( *x ) ).

union( set( *x ), set( *y ), set( *y ) ) <-
    ATOM( *x )
  & element( *x, set( *y ) ).

union( set( *x ), set( *y ), set( *x & *y ) ) <-
    ATOM( *x )
  & ~element( *x, set( *y ) ).

union( set( *x & *y ), set( *x & *z ), set( *x & *w ) ) <-
    union( set( *y ), set( *z ), set( *w ) ).

union( set( *x & *y ), set( *z & *w ), set( *x & *v ) ) <-
    ~EQ( *x, *z )
  & ~element( *x, set( *w ) )
  & union( set( *y ), set( *z & *w ), set( *v ) ).

union( set( *x & *y ), set( *z & *w ), set( *v ) ) <-
    ~EQ( *x, *z )
  & element( *x, set( *w ) )
  & union( set( *y ), set( *z & *w ), set( *v ) ).

/*
*/

difference( set( NIL ), *x, set( NIL ) ).
difference( set( *x ), set( NIL ), set( *x ) ).

difference( set( *x ), set( *x ), set( NIL ) ).

difference( set( *x ), set( *y ), set( *x ) ) <-
    intersection( set( *x ), set( *y ), set( NIL ) ).

difference( set( *x & *y ), set( *x ), set( *y ) ).

difference( set( *x & *y ), set( *y ), set( *x ) ).

difference( set( *x & *y ), set( *x & *v ), set( *w ) ) <-
    difference( set( *y ), set( *v ), set( *w ) ).

difference( set( *x & *y ), set( *z & *w ), set( *x & *v ) ) <-
    ~EQ( *x, *z )
  & ~element( *x, set( *w ) )
  & difference( set( *y ), set( *z & *w ), set( *v ) ).

difference( set( *x & *y ), set( *z & *w ), set( *v ) ) <-
    ~EQ( *x, *z )
  & element( *x, set( *w ) )
  & difference( set( *y ), set( *z & *w ), set( *v ) ).

/*
*/

remove( *x, set( *x ), set( NIL ) ).

remove( *x, set( *x & *y ), set( *y ) ).

remove( *x, set( *y & *x ), set( *y ) ).

remove( *x, set( *y & *rest ), set( *y & *z ) ) <-
    remove( *x, set( *rest ), set( *z ) ).
```

```
/*USAGE

  RELATION: sort   _sort
         sortb

  DESCRIPTION: sorts DLOG sets in lexicographic order.

  NEEDS:

  USED IN: eqset     SETFNS1


*/

_sort( *x.NIL, *x.NIL ).

_sort( *x.*y.NIL, *x.*y.NIL ) <-
   ATOM( *x )
  & ATOM( *y )
  & LE( *x, *y ).

_sort( *x.*y.NIL, *y.*x.NIL ) <-
   ATOM( *x )
  & ATOM( *y )
  & LE( *y, *x ).

_sort( *x.*rest, *x.*y.*z ) <-
   _sort( *rest, *y.*z )
  & LE( *x, *y )
  & /.

_sort( *x.*rest, *y.*v ) <-
   _sort( *rest, *y.*z )
  & LE( *y, *x )
  & _sort( *x.*z, *v )
  & /.

/*
*/

sort( *x, *x ) <-
   ATOM( *x ).

sort( *x & *y, *x & *y ) <-
   ATOM( *x )
  & ATOM( *y )
  & LE( *x, *y ).

sort( *x & *y, *y & *x ) <-
   ATOM( *x )
  & ATOM( *y )
  & LE( *y, *x ).
```

```
sort( *x & *rest, *x & *y & *z ) <-
    sort( *rest, *y & *z )
  & LE( *x, *y ).

sort( *x & *rest, *y & *v ) <-
    sort( *rest, *y & *z )
  & LE( *y, *x )
  & sort( *x & *z, *v ).

/*
*/

sortb( *x, *x, *base ) <-
    apply( *base, *x ).

sortb( *x & *y, *x & *y, *base ) <-
    apply( *base, *x )
  & apply( *base, *y )
  & ordered( *x, *y ).

sortb( *x & *y, *y & *x, *base ) <-
    apply( *base, *x )
  & apply( *base, *y )
  & ordered( *y, *x ).

sortb( *x & *rest, *x & *y & *z, *base ) <-
    sortb( *rest, *y & *z, *base )
  & ordered( *x, *y )
  & /.

sortb( *x & *rest, *y & *z, *base ) <-
    sortb( *rest, *y & *v, *base )
  & ordered( *y, *x )
  & sortb( *x & *v, *z, *base )
  & /.


/*
*/

_sortb( *x.NIL, *x.NIL, *base ) <-
    apply( *base, *x ).

_sortb( *x.*y, *x.*y, *base ) <-
    apply( *base, *x )
  & apply( *base, *y )
  & ordered( *x, *y ).

_sortb( *x.*y, *y.*x, *base ) <-
    apply( *base, *x )
  & apply( *base, *y )
  & ordered( *y, *x ).
```

```
_sortb( *x.*rest, *x.*y.*z, *base ) <-
    _sortb( *rest, *y.*z, *base )
  & ordered( *x, *y )
  & /.

_sortb( *x.*rest, *y.*z, *base ) <-
    _sortb( *rest, *y.*v, *base )
  & ordered( *y, *x )
  & _sortb( *x.*v, *z, *base )
  & /.

/*
*/

ordered( *x, *y ) <-
    ATOM( *x )
  & ATOM( *y )
  & LE( *x, *y ).

ordered( *x.*r1, *y.*r2 ) <-
    ATOM( *x )
  & ATOM( *y )
  & LT( *x, *y )
  | EQ( *x, *y ) & ordered( *r1, *r2 ).

ordered( *x & *r1, *y & *r2 ) <-
    ATOM( *x )
  & ATOM( *y )
  & LT( *x, *y )
  | EQ( *x, *y ) & ordered( *r1, *r2 ).
```

```
/*USAGE

 RELATION: start

 DESCRIPTION: Invokes DLOG 1.0.

 NEEDS: input      INPUT1

 USED IN:

*/

START <-
   start.

start <-
   print( 'DLOG 1.0' )
  & restart.

start.

ERROR <-
   print( 'Re-enter last input' )
  & RETRY( restart ).

restart <-
   NEWLINE
  & READ( *input )
  & input( *input )
  & NEWLINE
  & RETRY( restart ).
```

```
/*USAGE

 RELATIONS: assertion
            implication
            constraint
            clause
            literal
            atom
            terms
            term
            individual
            set
            lambda
            individual_variable
            individual_constant
            individual_description
            definite_individual
            indefinite_individual
            set_variable
            set_constant
            set_description
            definite_set
            indefinite_set
            aggregate
            check_freevar
            check_term
            check_input

 DESCRIPTION: Grammar for DLOG assertions. Each unary predicate
      will verify the syntax of the corresponding DLOG object.
      The check_input predicate provides a hook for the
      input parser to apply transformations on objects of that
      type. If "input_mode" is on, then any relevant constraints
      are applied, otherwise the syntax derivation proceeds as
      specified.

 NEEDS: deterministic UTILS1
      bind

 USED IN: parse      PARSE1

*/

assertion( *s ) <-
   atom( *s )
  | implication( *s )
  | constraint( *s ).

/*
*/

implication( *c <- *a ) <-
   atom( *c )
```

```
    & clause( *a ).

 /*
 */

  constraint( *a <= *c ) <-
      atom( *a )
    & clause( *c ).

 /*
 */

  clause( *l ) <-
      literal( *l ).

  clause( *l & *rest ) <-
      ~individual_variable( *l )
    & clause( *l )
    & clause( *rest ).

  clause( *l | *rest ) <-
      ~individual_variable( *l )
    & clause( *l )
    & clause( *rest ).

  clause( *l ! *rest ) <-
      ~individual_variable( *l )
    & clause( *l )
    & clause( *rest ).

 /*
 */

  literal( ~*l ) <-
      atom( *l ).

  literal( *l ) <-
      atom( *l ).

 /*
 */

  atom( *a ) <-
      SKEL( *a )
    & CONS( *p.*tlist, *a )
    & ~OP( *p, *, * )
    & terms( *tlist ).

 /*
 */

  terms( *t.NIL ) <-
      deterministic( term( *t ) ).
 terms( *t.*tlist ) <-
     ¬bind( *tlist, NIL )
   & term( *t )
   & terms( *tlist ).

/*
*/

 term( *t ) <-
     deterministic( individual( *t ) | set( *t ) | lambda( *t ) ).

/*
*/

 individual( *i ) <-
     deterministic( individual_variable( *i ) )
   | individual_constant( *i )
   | individual_description( *i ).

/*
*/

 set( *s ) <-
     set_variable( *s )
   | set_constant( *s )
   | set_description( *s ).

/*
*/

 lambda( lambda( *x, *expr ) ) <-
     individual_variable( *x )
   & clause( *expr )
   & check_freevar( *x, *expr ).

/*
*/

 individual_variable( *v ) <-
     VAR( *v ).

/*
*/

 individual_constant( *c ) <-
     INT( *c )
   | ATOM( *c ).

/*
*/

 individual_description( *i ) <-
```

```
    definite_individual( *i )
  | indefinite_individual( *i ).

/*
*/

 definite_individual( the( *x, *e ) ) <-
     individual_variable( *x )
   & clause( *e )
   & check_freevar( *x, *e )
   & /
   & check_input( the( *x, *e ) ).

/*
*/

 indefinite_individual( an( *x, *e ) ) <-
     individual_variable( *x )
   & clause( *e )
   & check_freevar( *x, *e ).

/*
*/

 set_variable( *s ) <-
     SKEL( *s )
   & CONS( set.*v.NIL, *s )
   & individual_variable( *v ).

/*
*/

 set_constant( set( *c ) ) <-
     individual_constant( *c ).

 set_constant( set( *c & *rest ) ) <-
     individual_constant( *c )
   & set_constant( set( *rest ) ).


/*
*/

 set_description( set( *s ) ) <-
     definite_set( set( *s ) ).

 set_description( set( *s ) ) <-
     indefinite_set( set( *s ) ).

/*
*/

 definite_set( set( *x : *e ) ) <-
```

```
    individual_variable( *x )
   & clause( *e )
   & check_freevar( *x, *e ).

/*
*/

 indefinite_set( set( *x ; set( *y ) : *e1 & *e2 ) ) <-
    individual_variable( *x )
   & individual_variable( *y )
   & clause( *e1 )
   & clause( *e2 )
   & check_freevar( *x, *e1 )
   & check_freevar( set( *y ), *e2 ).

 indefinite_set( set( *ag ) ) <-
    aggregate( *ag ).

/*
*/

 aggregate( *x ) <-
    individual_variable( *x )
   & /
   & FAIL.

 aggregate( *a1 | *a2 ) <-
    aggregate_1( *a1 )
   & aggregate_1( *a2 ).

 aggregate( *a1 ! *a2 ) <-
    aggregate_1( *a1 )
   & aggregate_1( *a2 ).

 aggregate( *x & *y ) <-
    aggregate_1( *x )
   & aggregate_2( *y ).

 aggregate( *x & *y ) <-
    aggregate_2( *x )
   & aggregate_1( *y ).

 aggregate_1( *c ) <-
    individual_constant( *c ).

 aggregate_1( *x | *y ) <-
    aggregate_1( *x )
   & aggregate_1( *y ).

 aggregate_1( *x ! *y ) <-
    aggregate_1( *x )
   & aggregate_1( *y ).
 aggregate_1( *x & *y ) <-
    aggregate_1( *x )
  & aggregate_1( *y ).

 aggregate_2( *x ! *y ) <-
    aggregate_1( *x )
  & aggregate_1( *y ).

 aggregate_2( *x | *y ) <-
    aggregate_1( *x )
  & aggregate_1( *y ).

/*
  utility relations for syntax checking
*/

 check_freevar( *x, *expr ) <-
    VAR( *x )
  & copy( *x.*expr, _.*z )
  & check_term( _, *z )
  & /.

 check_freevar( set( *x ) , *expr ) <-
    VAR( *x )
  & copy( *x.*expr, _.*z )
  & check_term( _, *z )
  & /.

 check_freevar( *x, *expr ) <-
    print( 'Warning: expecting free variable'''.*x.''' in '''.*expr.'''' ).

/*
*/

 check_term( *x, *y ) <-
    SKEL( *y )
  & CONS( *p.*tlist, *y )
  & check_term_1( *x, *tlist ).

 check_term( *x, *y ) <-
    ATOM( *y )
  & EQ( *x, *y ).

 check_term_1( *x, *y.*rest ) <-
    check_term( *x, *y )
  | check_term_1( *x, *rest ).

/*
*/

 check_input( * ) <-
    ~input_mode
  & /.
```

```
check_input( *t ) <-
    input_constraint( *t ).
```

```
/*USAGE

  RELATION: transaction

  DESCRIPTION: reads and processes a DLOG/CMS transaction.

  NEEDS: assertion   SYNTAX1
        verify_ic   IC1
        user        UTILS1
        augment
        print
        read_skeleton
        update

  USED IN: parse  PARSE1

*/

transaction <-
   print( 'Begin transaction' )
  & NEWLINE
  & ADDAX( _trans( NIL ) )
  & ADDAX( _tno( 0 ) )
  & read_trans
  & process_trans
  & DELAX( _trans( * ) )
  & DELAX( _tno( * ) ).

/*
*/

read_trans <-
   read_skeleton( *t )
  & read_trans_1( *t )
  & /.

read_trans_1( end ).

read_trans_1( list ) <-
   list_trans
  & RETRY( read_trans ).

read_trans_1( *t ) <-
   assertion( *t )
  & update_trans( *t )
  & assert_trans( *t )
  & RETRY( read_trans ).

read_trans_1( *t ) <-
   print( 'Illegal syntax in '.*t )
  & print( 'Input ignored' )
  & RETRY( read_trans ).
```

```
/*
*/

 process_trans <-
    _trans( NIL )
   & print( 'Null transaction...processing terminates' ).

 process_trans <-
    _trans( *t )
   & process_trans_1( *t )
   & user( 'Update DB with transaction?' )
   & print( 'Transaction completed' ).

 process_trans <-
    _trans( *t )
   & revoke_trans( *t )
   & print( 'Transaction revoked...processing terminates' ).

 process_trans_1( NIL ).

 process_trans_1( ( *tno.*t ).*rest ) <-
    verify_ic( *t )
   & process_trans_1( *rest ).

/*
*/

 assert_trans( *a <= *c ) <-
    print( 'Assert constraint: '.*a.'<='.*c )
   & ADDAX( *a <= *c ).

 assert_trans( *c <- *a ) <-
    print( 'Assert implication: '.*c.'<-'.*a )
   & ADDAX( *c <- *a ).

 assert_trans( *a ) <-
    print( 'Assert atom: '.*a )
   & ADDAX( *a ).

/*
*/

 revoke_trans( NIL ).

 revoke_trans( ( *tno.*t ).*rest ) <-
    DELAX( *t )
   & revoke_trans( *rest ).

/*
*/

 update_trans( *new ) <-
    _trans( *t )
    & _tno( *tno )
    & SUM( 1, *tno, *ntno )
    & augment( *t, *ntno.*new, *nt )
    & update( _trans( *t ), _trans( *nt ) )
    & update( _tno( *tno ), _tno( *ntno ) ).

/*
*/

list_trans <-
    _trans( NIL )
  & print( 'Transaction is empty' ).

list_trans <-
    _trans( *t )
  & list_trans_1( *t ).

list_trans_1( NIL ).

list_trans_1( (*tno.*t).*rest ) <-
    print( *tno.': '.*t )
  & list_trans_1( *rest ).
```

```
/*USAGE

 RELATION: unify

 DESCRIPTION: unifies two DLOG literals, the first a query literal,
      the second a DB literal. The atom "heuristic_mode" is
      a global switch which enables the heuristic clauses.

 NEEDS: _element    SETFNS1
       _extension    EXT1
       local_ctx    CTX1
       _all        ALL1
       eqlambda    META1
       eqset
       apply
       extends
       individual_variable  SYNTAX1
       individual_constant
       individual
       set
       indefinite_set
       debug    UTILS1
       print

 USED IN: derivable    DERIVE1
        extends    META1

*/

 unify( *qlit, *dblit ) <-
    CONS( *p.*qterms, *qlit )
  & debug( unify, 'Q: '.*qlit.' DB: '.*dblit )
  & CONS( *p.*dbterms, *dblit )
  & unify_1( *qterms, *dbterms )
  & /.

/*
*/

 unify_1( NIL, NIL ).

 unify_1( *c1.*qts, *c2.*dbts ) <-  /* for efficiency only */
    individual_constant( *c1 )
  & individual_constant( *c2 )
  & ~bind( *c1, *c2 )
  & /( unify( *, * ) )
  & FAIL.

 unify_1( *i1.*qts, *i2.*dbts ) <-
    individual( *i1 )
  & individual( *i2 )
  & bind( *i1, *i2 )
  & unify_1( *qts, *dbts ).
```

```
unify_1( *s1.*qts, *s2.*dbts ) <-
   set( *s1 )
  & set( *s2 )
  & bind( *s1, *s2 )
  & unify_1( *qts, *dbts ).

unify_1( the( *x, *e ).*qts, *c.*dbts ) <-
   individual_constant( *c )
  & _extension( set( *x : *e ), *c.NIL )
  & unify_1( *qts, *dbts ).

unify_1( the( *x, *e1 ).*qts, an( *y, *e2 ).*dbts ) <-
   eqlambda( lambda( *x, *e1 ), lambda( *y, *e2 ) )
  & _extension( set( *y : *e2 ), *z.NIL )
  & unify_1( *qts, *dbts ).

unify_1( an( *v, *e ).*qts, *c.*dbts ) <-
   individual_constant( *c )
  & apply( lambda( *v, *e ), *c )
  & unify_1( *qts, *dbts ).

unify_1( *c.*qts, an( *v, *e ).*dbts ) <-
   individual_constant( *c )
  & apply( lambda( *v, *e ), *c )
  & unify_1( *qts, *dbts ).

unify_1( an( *v1, *e1 ).*qts, an( *v2, *e2 ).*dbts ) <-
   eqlambda( lambda( *v1, *e1 ), lambda( *v2, *e2 ) )
  & unify_1( *qts, *dbts ).

unify_1( *s1.*qts, *s2.*dbts ) <-
   set( *s1 )
  & set( *s2 )
  & eqset( set( *s1 ), set( *s2 ) )
  & unify_1( *qts, *dbts ).


unify_1( lambda( *v, *e1 ).*qts, lambda( *v, *e2 ).*dbts ) <-  /* heuristic */
   heuristic_mode
  & query_mode
  & extends( *e2, *e1 )
  & print( 'Heuristic assumption: '.*e1.' extends '.*e2 )
  & unify_1( *qts, *dbts ).

unify_1( lambda( *v, *e1 ).*qts, lambda( *v, *e2 ).*dbts ) <-
   eqlambda( lambda( *v, *e1 ), lambda( *v, *e2 ) )
  & unify_1( *qts, *dbts ).
```

```
/*USAGE

 RELATION: user
        user_select
        print
        printl
        prompt
        augment
        acknowledge
        concatenate
        copy
        bind
        list
        gensym
        letter_prefix
        digit_suffix
        nth_digit
        nth_char
        length
        deterministic
        atomic
        menu_value
        menu_position
        menu_print
        menu_pick
        nth_element
        read_integer
        read_string
        read_skeleton
        read_charlist
        reverse
        append
        for
        lc_UC
        load
        cons_amp
        amp_cons
        bar_cons
        bang_cons
        disjunct
        debug

DESCRIPTION: various utilities used in many DLOG subsystems.

NEEDS: OPS1

USED IN: almost everywhere.

*/

/*
 user poses yes/no queries to the user
*/
```

```
user( *list ) <-
    print( *list )
  & acknowledge.

/*
  uses menu_value to have user pick one element of a list
  formed with &, |, or !
*/

user_select( *x & *y, *s ) <-
    amp_cons( *x & *y, *list )
  & menu_value( *list, *s ).


user_select( *x | *y, *s ) <-
    bar_cons( *x | *y, *list )
  & menu_value( *list, *s ).


user_select( *x ! *y, *s ) <-
    bang_cons( *x ! *y, *list )
  & menu_value( *list, *s ).

/*
  print writes a list of arguments
*/

print( *s1.*list ) <-
    WRITECH( *s1 )
  & /
  & print( *list ).

print( *s ) <-
    WRITECH( *s )
  & NEWLINE.

/*
  printl prints the elements of a list *x.*y. ... .*z.NIL
*/

printl( *x.NIL ) <-
    WRITECH( *x )
  & NEWLINE
  & /.

printl( *x.*rest ) <-
    WRITECH( *x )
  & WRITECH( ' ' )
  & printl( *rest ).

/*
  prompt is identical to print, except '&' carriage control is used
```

```
   NOTE: PROLOG/CMS doesn't support CONTROL( CC, * ) switch (yet).
 */

 prompt( *list ) <-
     CONTROL( CC, *sw )
   & ADDAX( CONTROL( CC, ON ) )
   & concatenate( ' ', *list, *prompt )
   & print( *prompt )
   & ADDAX( CONTROL( CC, *sw ) ).

/*
  augment is true when the last arg is a copy of the first list arg
  with the second arg at the end.
*/

 augment( NIL, *x, *x.NIL ).

 augment( *x.NIL, *y, *x.*y.NIL ).

 augment( *u.*x, *y, *u.*z ) <-
     augment( *x, *y, *z ).

/*
  acknowledge asks the user to respond 'y' or 'n'
*/

 acknowledge <-
     prompt( 'Acknowledge(y|n):' )
   & read_string( *s )
   & STRING( *s, y.* ).

/*
  concatenate put two strings together.
*/

concatenate( *id1, *id2, *id1id2 ) <-
     STRING( *id1, *id1list )
   & STRING( *id2, *id2list )
   & combine( *id1list, *id2list, *id1id2list )
   & /
   & STRING( *id1id2, *id1id2list ).

combine( *x.NIL, *y, *x.*y ).

combine( *x.*rest, *y, *z ) <-
     combine( *rest, *y, *u )
   & combine( *x.NIL, *u, *z ).

/*
*/

 copy( *x, *y ) <-
     ADDAX( _copy( *x ) )
  & DELAX( _copy( *y ) ).

/*
*/

bind( *x, *x ).

/*
*/

list( *x.NIL ) <-
   ATOM( *x ).

list( *x.*y ) <-
   ATOM( *x )
  & list( *y ).

/*
*/

symbol_counter( 0 ).  /* intialize counter */

gensym( *pfx, *unique ) <-
   symbol_counter( *new )
  & DELAX( symbol_counter( * ) )
  & SUM( *new, 1, *next )
  & ADDAX( symbol_counter( *next ) )
  & concatenate( *pfx, *new, *unique ).

/*
*/

letter_prefix( *x, *y ) <-
   STRING( *y, *list )
  & lp( *pfxlist, *list )
  & STRING( *x, *pfxlist ).

lp( *x.NIL, *x.NIL ) <-
   LETTER( *x ).

lp( *x.NIL, *x.*y.*list ) <-
   LETTER( *x )
  & DIGIT( *y ).

lp( *x.*y, *x.*list ) <-
   LETTER( *x ) & lp( *y, *list ).

/*
*/

digit_suffix( *x, *y ) <-
   STRING( *y, *list )
  & ds( *sfxlist, *list )
  & STRING( *x, *sfxlist ).

 ds( *x.NIL, *x.NIL ) <-
    DIGIT( *x ).

 ds( *x.*y, *x.*list ) <-
    DIGIT( *x )
   & ds( *y, *list ).

 ds( *x, *y.*list ) <-
    LETTER( *y )
   & ds( *x, *list).

/*
*/

 nth_digit( *n, *int, *d ) <-
    INT( *int )
   & INT( *n )
   & STRING( *int, *digitlist )
   & length( *digitlist, *l )
   & LE( *n, *l )
   & GE( *n, 0 )
   & nth_char_1( *n, *digitlist, *d ).

/*
*/

 nth_char( *n, *str, *c ) <-
    INT( *n )
   & ATOM( *str )
   & STRING( *str, *charlist )
   & length( *charlist, *l )
   & LE( *n, *l )
   & GT( *n, 0 )
   & nth_char_1( *n, *charlist, *c ).

 nth_char_1( 1, *c.*rest, *c ).

 nth_char_1( *n, *x.*rest, *c ) <-
    DIFF( *n, 1, *n1 )
   & nth_char_1( *n1, *rest, *c ).

/*
*/

 length( NIL, 0 ).

 length( *x.*y, *n ) <-
    length( *y, *n1 )
   & SUM( *n1, 1, *n ).

/*
```

```
*/

 deterministic( *g ) <-
    *g
   & /.

/*
*/

 atomic( *a ) <-
    CONS( *p.*, *a )
   & ~OP( *p, *, * ).

/*
*/

 menu_value( *list, *val ) <-
    menu_position( *list, *n )
   & nth_element( *n, *list, *val ).

/*
*/

 menu_position( *list, *n ) <-
    NEWLINE
   & menu_print( *list, 1 )
   & NEWLINE
   & length( *list, *max )
   & menu_pick( *n, 1, *max )
   & /.

/*
*/

 menu_print( NIL, * ).

 menu_print( *e.*rest, *n ) <-
    print( *n.': '.*e )
   & SUM( *n, 1, *n1 )
   & menu_print( *rest, *n1 ).

/*
*/

 menu_pick( *n, *min, *max ) <-
    prompt( 'Selection?' )
   & read_integer( *n )
   & GE( *n, *min )
   & LE( *n, *max ).


 menu_pick( *, *, * ) <-
    print( 'Bad selection' )
   & RETRY( menu_pick( *, *, * ) ).

/*
*/

 nth_element( 1, *e.*rest, *e ).

 nth_element( *n, *x.*rest, *e ) <-
     INT( *n )
   & DIFF( *n, 1, *n1 )
   & nth_element( *n1, *rest, *e ).

/*
*/

 read_integer( *n ) <-
     read_charlist( *l )
   & STRING( *n, *l )
   & INT( *n )
   & /.

 read_integer( *n ) <-
     print( 'Illegal integer' )
   & FAIL.

/*
*/

 read_string( *s ) <-
     read_charlist( *l )
   & STRING( *s, *l )
   & /.

 read_string( * ) <-
     print( 'Illegal string' )
   & FAIL.

/*
*/

 read_skeleton( *sk ) <-
     READ( *sk )
   & /.

/*
*/


 read_charlist( *l ) <-
     ADDAX( _cl( NIL ) )
   & read_charlist_1
   & /
   & DELAX( _cl( *lr ) )
```

```
  & reverse( *lr, *l ).

 read_charlist_1 <-
    READCH( *c )
  & update( _cl( *cur, _cl( *c.*cur ) )
  & EOL.

/*
*/

 reverse( *x.NIL, *x.NIL ).

 reverse( *x.*y, *z ) <-
    reverse( *y, *w )
  & append( *w, *x.NIL, *z ).

/*
*/

 append( NIL, *x, *x ).

 append( *u.*x, *y, *u.*z ) <-
    append( *x, *y, *z ).

/*
*/

 for( *b, *e, *i, *g ) <-
    ADDAX( _for( *b.*e.*i.*g ) )
  & forgoal
  & DELAX( _for( * ) ).

 for( *, *, *, * ) <-
    DELAX( _for( * ) )
  & FAIL.

 forgoal <-
    _for( *b.*e.*i.*g )
  & LE( *b, *e )
  & /
  & copy( *g, *cg )
  & *cg
  & SUM( *b, *i, *nb )
  & update( _for( *b.*e.*i.*g ), _for( *nb.*e.*i.*g ) )
  & RETRY( forgoal ).

 forgoal.

/*
*/

 update( *o, *n ) <-
    DELAX( *o )
```

```
   & ADDAX( *n ).

 /*
 */

 lc_UC( *lc, *uc ) <-
     STRING( *lc, *lc1 )
   & lc_UC_1( *lc1, *uc1 )
   & STRING( *uc, *uc1 ).

 lc_UC_1( NIL, NIL ).

 lc_UC_1( *lc.*rlc, *uc.*ruc ) <-
     UPSHIFT( *lc, *uc )
   & lc_UC_1( *rlc, *ruc ).

 lc_UC_1( *lc.*rlc, *lc.*ruc ) <-
     lc_UC_1( *rlc, *ruc ).

 /*
 */

 load <-
     prompt( 'Filename ( q to quit )? ')
   & read_string( *f )
   & ~EQ( *f, q )
   & lc_UC( *f, *ucf )
   & /
   & LOAD( *ucf )
   & print( *ucf.' loaded' )
   & RETRY( load ).

 load.

 /*
 */

 cons_amp( NIL, NIL ).

 cons_amp( *x.NIL, *x ).

 cons_amp( *x.*y, *x & *rest ) <-
     SKEL( *y )
   & cons_amp( *y, *rest ).

 /*
 */

 amp_cons( *x, *x.NIL ) <-
     ~CONS( &.*, *x ).

 amp_cons( *x & *y, *x.*z ) <-
     ~CONS( &.*, *x )
   & amp_cons( *y, *z ).

/*
*/

 bar_cons( *x, *x.NIL ) <-
     ~CONS( |.*, *x ).

 bar_cons( *x | *y, *x.*z ) <-
     ~CONS( |.*, *x )
   & bar_cons( *y, *z ).

/*
*/

 bang_cons( *x, *x.NIL ) <-
     ~CONS( !.*, *x ).

 bang_cons( *x ! *y, *x.*z ) <-
     ~CONS( !.*, *x )
   & bang_cons( *y, *z ).

/*
*/

 disjunct( *dj, *dj ) <-
     CONS( *p.*, *dj )
   & ~OP( *p, *, * ).

 disjunct( *dj, *dj | *rest ).

 disjunct( *dj, * | *rest ) <-
     disjunct( *dj, *rest ).

/*
*/

 debug( *n, *g ) <-
     debug( *n )
   & print( *n.': '.*g ).

 debug( *, * ).
```

Appendix 2
Departmental Database predicate descriptions

The following list includes a brief description of the predicates that are used to define the Department Data Base (DDB) application data base.

student_program_contribution(x,y,z). The course z will fulfill some requirement toward the completion of program y for student x. This predicate is derivable when the course z has not yet been taken by student x, but if completed, would make a contribution to the completion of program y.

completed(x,y). Student x has completed the course y. Matriculation courses do not require that the student had first been enrolled; nor do they require that a grade be recorded. For non-matriculation courses, the student x must have been enrolled in course y, and must have attained a passing grade in y before this relation can be asserted.

program_enrolled(x,y). This asserts that student x is enrolled in degree program y. It requires that there be some program in which x has successfully registered.

registered(x,y,z). This asserts that student x is registered in year z of degree program x. This assertion cannot be made unless the student in question has satisfied the prerequisites for year z of program y.

elective(x,y). This predicate asserts that course y is a legal elective for program y. It is a weaker assertion than *faculty_elective*(x,y,z), since it does not specify which faculty the course is an elective for, nor what faculty the course is presumed to be from (see below).

dept_program_prereq(u,v,w,x,y,z). This predicate asserts that course z is a requirement in year y, for any degree program offered in department u, at level v (e.g., Bachelor, Master, etc.), in stream w (e.g., majors, honours, etc.), with field x (e.g., Computer Science, Physics-Mathematics, etc.). Requirements stated in this way provide the details of courses specified at the department level, and can be inherited by program course requirements.

matriculation_course(x). This specifies the known domain of matriculation courses (e.g., ALGEBRA11, ALGEBRA12, CHEM11, etc.).

course_enrolled(x,y). This predicate asserts that student x is enrolled in course y. The predication cannot be made without verifying that the student has first satisfied the necessary prerequisites for course y, and furthermore, that course y is a contribution to the student's program (an admittedly fascist constraint).

field_of(x,y). This predicate asserts that the field of degree program x is field y. For example the field of "BScHonoursCSMATH" is "Computer Science and Mathematics."

year_of(x,y). This predicate asserts that y is the year in which student x is currently enrolled. This is derivabled from the *registered*(x,y,z) predicate, and therefore carries the same constraints.

recommended(x,y,z). This predicate is used to record the recommendations for filling elective courses in degree program requirements. It means that course z is recommended as an elective in year y of program x.

faculty_of(x,y). This predicate asserts that the faculty of x is y. The value of x can be a course, department or degree program.

course(x). This predicate specifies the domain of courses; the only constraint is that each course name x be a string of the form "<D><N>" where "<D>" is a known department, and "<N>" is an integer.

program_of(x,y). This predicate asserts that student x is enrolled in degree program y. It is identical to the predicate *program_enrolled*(x,y).

program_req(x,y,z). This predicate asserts that z is a requirement of year y for degree program x.

stream_of(x,y). This predicate asserts that the stream of degree program x is y (e.g., majors, honours, etc.).

level(x). This predicate specifies the domain of degree program levels (e.g., Bachelor, Master, Doctorate).

year(x). This predicate specifies the domain of degree program years. In the current DDB application data base, this includes "first," "second," "third," and "fourth."

faculty_grad_req(w,x,y,z). This predicate asserts that z is a graduation requirement for all degree programs from faculty w at level x (e.g., bachelor, master, etc.), in stream y (e.g., majors, honours, etc.). The set of faculty graduation requirements are a subset of the extension of the more general predicate *graduation_req*(x,y) for appropriate programs x.

program_prereq(x,y,z). This predicate asserts that requirement z is a prerequisite for enrolment in year y of program x.

faculty_elective(x,y,z). This predicate asserts that course z can be considered to be a course from faculty y, when considered as an elective for programs offered in the faculty x.

unit_value(x,y). This predicate asserts that course x has unit value y

faculty_course_req(v,w,x,y,z). This predicate is necessary to specify course requirements set at the faculty level (cf. *dept_course_req*). The predicate asserts that course z is a requirement for all degree programs with year y, in stream x (e.g., majors, honours, etc.), at level w (e.g., bachelor, master, etc.) in faculty v. The requirements specified at this level are a subset of those specified by the predicate *course_req*(x,y) for the appropriate degree programs.

course_equivalent(x,y). This predicate asserts that course x is viewed as equivalent to course y. This is used for cross listed courses, or for those which are similar enough so that credit cannot be had for both.

faculty_program_req(v,w,x,y,z). This predicate is used to specify general program requirements set at the faculty level (e.g., a certain number of Science units). It asserts that requirement z must be satisfied for year y of all programs with stream x, level w, and faculty v.

faculty_program_prereq(v,w,x,y,z). This predicate is used to specify faculty level requirements z that are prerequisite to enrolling in year y of a degree program with stream x, at level w in faculty v. As for *dept_program_prereq*, these prerequisites are a subset of those specified by *program_prereq* for the appropriate degree programs.

program_contribution(x,y,z). This predicate is true when course z will make a contribution toward the requirements of completing year y of the degree program in which student x is enrolled. The derivation of this predicate requires the use of the DLOG predicate *extends* (see §5.4).

eligible_for_degree(x,y). This predicate is true when student x has satisfied all the graduation requirements for degree program y. Derivation of this relation as a query initiates the most complex and time consuming computation possible in the version of DLOG in which this application data base was developed.

eligible_for_course(x,y). This predicate is true when student x has satisfied the prerequisites for course y.

eligible_for_admission(x,y). This predicate is true when person x (i.e., someone know only by *name_id* who is not yet a student) is known to have completed the prerequisites for admission to degree program y.

eligible_for_year(x,y). This predicate is true when student x has completed the prerequisites for admission to year y of the program they are currently enrolled in.

field(x). This predicate specifies the domain of known fields (e.g., Computer Science, Computer Science and Mathematics, Computer Science and Physics, etc.).

dept_of(x,y). This predicate specifies that the department of course x is department y.

faculty(x). This predicate specifies the domain of known faculties (e.g., Science, Arts, etc.).

course_no(x,y). This predicate asserts that the course number of course x is the integer y.

program(x). This predicate specifies the domain of known degree programs (e.g., BScMajorsCS, BScHonoursCS, etc.).

grad_req(x,y). This predicate asserts that requirement y is one requirement to be satisfied in order to graduate with degree x.

level_of(x,y). This predicate asserts that the level (e.g., bachelor, master, etc.) of degree program x is y.

stream(x). This predicate specifies the domain of known streams (e.g., majors, honours, etc.).

course_req(x,y,z). This predicate asserts that course z is a requirement of year y for program z. In this application data base, requirements of this sort are normally stated as lambda expressions (see §4.1.2).

head(x). This predicate specifies the domain of known department heads. Its assertion requires that the x in question be a faculty member, and the member of some department (i.e., $\exists y.depart\_of(x,y)$).

dept(x). This predicate specifies the domain of know departments (e.g., CS, MATH, ENGL, etc.).

previous_year(x,y). This predicate specifies a total ordering on the four years of undergraduate degree programs, i.e., that "first_year""second_year""third_year""fourth_year."

faculty_member(x). This predicate specifies the domain of known faculty members. In this application data base, it is required only when admission to some course requires "the head's permission" in which case the existence of a department head presupposes that he/she is a faculty member. One elaboration of this application data base would be to provide the details of course lecturers.

course_prereq(x,y). This predicate is used to assert that course x requires the satisfaction of requirements y. In this axiomatization of the application domain, these requirements are specified as lambda expressions to be satisfied (see §4.1.2).

student(x). This predicate specifies the domain of known students. Notice that, for this application domain, the successful specification of a new student transcript will update this relation (see §5.3.1).

grade_of(x,y,z). This predicate asserts that the grade attained by student x in course y is z.

degree(x). This predicate specifies the domain of known degree program names.

passing_grade_of(x,y). This predicate specifies that the passing grade of course x is the grade y. Currently in this application data base, the value for all courses is 50.

age_of(x,y). This predicate asserts that the age of person x is y years.

name_id(x). This predicate specifies the domain of known person names. The intended interpretation is that unique name identifiers correspond to unique persons in the application domain model. Both *name_id* and *student* are necessary, because the current application domain can hold assertions about a person who exists, but has not yet been enrolled as a student.

Appendix 3
KRL descriptors in logic

The KRL language of Bobrow and Winograd [Bobrow77a, Bobrow77b, Bobrow79] represents an ambitious effort to embody current ideas about knowledge representation (e.g., frames [Minsky75]) into a comprehensive computational framework. Much has been written in response to KRL, but most of the reactions express confusion about the possible contributions (e.g., [Lehnert78]), berates the language for lack of clarity (e.g., [McDermott78]), or argues that much of the language's compendium of concepts is directly interpretable via the denotational semantics of first order logic (e.g., [Hayes80]).

When the concepts of epistemological and heuristic adequacy were articulated by McCarthy and Hayes [McCarthy69, McCarthy77], they argued that much work in Artificial Intelligence had addressed heuristic issues at the expense of epistemological ones, and that progress in AI required further understanding of the latter. Hayes' analysis of frames and KRL [Hayes80] uses Tarskian semantics as a normative theory of meaning to argue that much of KRL's machinery is heuristic in flavour, and that the non-Tarskian excesses (self reference and non-monotonicity) are the only interesting epistemological aspects.

While Hayes' analysis is a useful antidote to KRL's "edifice of notation,"[40] the experience with constructing DLOG suggests that there may be some epistemological import to the apparently heuristic aspects of KRL: in particular, it seems that descriptions are not merely abbreviations for their contextual definitions, but that they are vital to symbolic reasoning in that they provide a method for packaging information in a form that would otherwise require much more extensive manipulation. (cf. abstract data types for programming languages). This analysis is speculative, but it seems plausible enough to warrant further investigation.

[40] [McDermott78, p. 280].

As Hayes [Hayes80] explains, the concept of a frame (or KRL unit) can be viewed as an epistemological notion if it can be argued that such structures are a basic component of knowledge — that they are included in our ontology. Here the contention is that the organization of knowledge into KRL units is the foundation for an AI theory of reasoning, not in spite of, but because of Hayes analysis. The intersting thing about this claim is not that it is new (e.g., it is the major reason Minsky [Minsky75] proposed the idea), but that the logical analysis in terms of descriptions supports the idea, and even suggests new directions for pursuing it.

Here we outline the framework of the idea. Bobrow and Winograd propose that most of KRL's reasoning is carried out by a comprehensive "matching" process, with little responsibility relegated to a "general purpose theorem prover."[41] But of course the DLOG pattern matcher is based on invocation of the DLOG proof procedure (cf. LOGLISP [Robinson82]). The most important idea supported by the logical analysis of descriptions is that the KRL style of reasoning is based on *demonstrating the equality of descriptions*. KRL acknowledges the importance of MERLIN's "mapping" procedure [Moore74], and that an object-centered representation should be constructed as collections of various forms of descriptors. The DLOG unification process, and especially the *extends* predicate (§5.4) demonstrates that this kind of reasoning can be interpreted logically, and that the "quality of the match"[42] can be based on logical notions, i.e., a partial derivation.

Here follows a rewriting of Hayes' rendering of various KRL examples [Hayes80]. Writing every KRL descriptor as a first order equality proposition demonstrates that the DLOG unification mechanism together with the *extends* meta predicate is the seed of an interesting theory of reasoning that seems to help resolve the long-standing dispute about whether reasoning should be based on logic.

KRL descriptors in logic

The language used to rewrite the various KRL examples is similar to that used in chapters 2, 3,

[41] [Bobrow77a, p. 24].

[42] [Bobrow77a, p. 26].

and 4. The various kinds of reasoning suggested by KRL would be performed by attempting to demonstrate the equality of descriptions as formulated in the following examples. Viewing some individual under one of possibly "multiple perspectives"[43] might be performed by demonstrating the equality of two individuals by instantiating one disjunct of a unit's descriptors written in disjunctive normal form. For example, the process of viewing "Juan" as a traveller is performed by attempting to match "Juan" with the KRL unit "G0043", and instantiating the disjunct beginning with the predication "Traveller(x)..."

direct pointer

KRL: Block17, PaloAlto, etc.

logic: any individual constant (i.e., non-logical constant).

perspective

KRL: (a Trip with destination=Boston Airline=TWA)

logic: $\epsilon x.\mathrm{trip}(x) \wedge \mathrm{destination}(x)=\mathrm{Boston} \wedge \mathrm{airline}(x)=\mathrm{TWA}$

specification

KRL: (the Actor from Act (a Chase with quarry={car22 (a Dodge)}))

logic: $\iota x.\mathrm{Actor}(x) \wedge x=\mathrm{actor\text{-}of}(\epsilon y.\mathrm{Act}(y) \wedge \mathrm{Chase}(y) \wedge \mathrm{quarry}(y)=\epsilon z.\mathrm{Dodge}(z) \wedge z=\mathrm{car22})$

predication

KRL: (which Owns (a Dog))

logic: $\epsilon x.\mathrm{Owns}(x,\epsilon y.\mathrm{Dog}(y))$

Using descriptions, an assertion would be

$\mathrm{Minnie}=\epsilon x.\mathrm{Owns}(x,\epsilon y.\mathrm{Dog}(y))$

Using a lambda expression (cf. [Hayes80]) we have

$\lambda x.(\exists y.\mathrm{Dog}(y) \wedge \mathrm{Owns}(x,y))\ \mathrm{Minnie}$

[43] [Bobrow77a, p. 6].

logical boolean

KRL: (OR (a Dog) {(a Cat) (which hasColor Brown)})

logic: $\epsilon y.[Dog(y) \vee [\epsilon y.cat(y) \wedge hasColor(y,Brown)]]$

restriction

KRL: (the {(a Mouse) (which Owns (a Dog))})

logic: $\iota x.Mouse(x) \wedge Owns(x,\epsilon y.Dog(y))$

selection

```
KRL: (using (the age for Person ThisOne)
       selectFrom (which isLessThan 2) Infant
              (which isAtLeast 12) Adult
       otherwise Child)
```

logic: $\iota x.[age(y)2 \wedge Infant(x)]$
$\vee [age(y) \geq 12 \wedge Adult(x)]$
$\vee [age(y)2 \wedge age(y) \leq 12 \wedge Child(x)]$

set specification

KRL: (SetOF {(an Integer) (which hasFactor 2)})

logic: $\epsilon X.[\forall x.x \in X \equiv integer(x) \wedge hasFactor(x,2)]$

KRL: (Items 2 4)

logic: $\epsilon X.[\forall x.x \in X \equiv [x=2 \vee x=4]]$

KRL: (Allitems 2 4 64 {(an Integer) (which hasFactor 3)})

logic: $\epsilon X.[\forall x.x \in X \equiv x=2 \vee x=4 \vee x=64$
$\vee x=\epsilon y.Integer(y) \wedge hasFactor(y,3)]$

KRL: (NotItems 51)

logic: $\epsilon X.51 \in X$

KRL: (In (SetOf {(an Integer) (Items 2 5 8) (NotItems 4)})

logic: $\epsilon x.[\exists X.x \in X \wedge \forall y.[y \in X \supset [x=2 \vee x=5$
$\vee x=8 \vee x=\epsilon z.[Integer(z) \wedge x=4]]]]$

contingency

```
KRL: during State24 then (the top from (a Stack with height=3)))

logic: ∈x.top(x,∈y.stack(y,State24)∧height(y,State24)=3)

KRL: during (a Dream with dreamer=Jacob) then (an Angel))

logic: ∈x.angel(x,∈y.state(y)∧y=∈z.dream(z)∧dreamer(z)=Jacob)
```

multiple perspectives

```
KRL: [G0043 UNIT Individual
      <SELF {(a Person with
             firstName="Juan"
             lastName={(a ForeignName)
                  (a String with firstCharacter="M"})
             age=(which IsGreaterThan 21))
            (a Traveller with
             preferredAirport=SJO
             age=Adult)
            (a Customer with
             credit=(a CreditCard with
                  company=UniversalCharge
                  number="G45-7923-220"))}>]

logic: G0043=∈x.individual(x)
         ∧[[Person(x)∧firstName(x)="Juan"
             ∧lastName=∈y.ForeignName(y)∧firstCharacter(y)="M"]
        ∨[Traveller(x)∧preferredAirport(x)=SJO
             ∧age(x)=Adult]
        ∨[Customer(x)∧credit(x)=∈y.CreditCard(y)
                                        ∧Company(y)=UniversalCharge
                                        ∧number(y)="G45-7923-220"]]
```

Appendix 4
Department Database implementation in DLOG

This appendix contains the current DLOG representation of the Department Database (DDB).

```
/*
 Department Data Base: Application command definitions


 The following PROLOG code is the User definition of the
 transcript command, as described in chapter 5, §5.3.1.

*/

user_command( transcript ).

/*
  Application command: transcript
*/

transcript <-
   menu_position( 'load'
                 .'save'
                 .'list'
                 .'edit'
                 .'create'
                 .'browse'
                 .NIL, *n )
  & transcript_1( *n )
  & user( 'Continue?' )
  & RETRY( transcript ).

transcript.

/*
*/

transcript_1( 1 ) <-
 load_transcript.

transcript_1( 2 ) <-
 save_transcript.

transcript_1( 3 ) <-
```

```
  list_transcript.

 transcript_1( 4 ) <-
  print( 'Not yet implemented' ).

 transcript_1( 5 ) <-
  create_transcript.

 transcript_1( 6 ) <-
  browse_predicate( name_id, 1 ).

/*
*/

 get_id( *n ) <-
    prompt( 'Student identifier? ' )
   & read_string( *n )
   & verify_id( *n )
   & /.

 verify_id( *n ) <-
    name_id( *n ).

 verify_id( *n ) <-
    user( 'Unknown name identifier...retry?' )
   & RETRY( get_id( * ) ).

/*
*/

 get_new_id( *n ) <-
    prompt( 'New name identifier? ' )
   & read_string( *n )
   & verify_new_id( *n )
   & /.

 verify_new_id( *n ) <-
    ~name_id( *n ).

 verify_new_id( *n ) <-
    user( 'Student id in use...retry?' )
   & RETRY( get_new_id( * ) ).

/*
*/

 get_program_id( *p ) <-
    prompt( 'Program name? ' )
   & read_string( *p )
   & verify_program( *p )
   & /.

 verify_program( *p ) <-
```

```
    program( *p ).

  verify_program( *p ) <-
     user( 'Unknown program...retry?' )
   & RETRY( get_program_id( * ) ).

/*
*/

 load_transcript <-
     get_id( *n )
   & lc_UC( *n, *nuc )
   & concatenate( 'T', *nuc, *f )
   & LOAD( *f )
   & print( *n.' transcript loaded' )
   & /.

 load_transcript <-
     print( 'transcript not obtainable' ).

/*
*/

 save_transcript <-
     get_id( *n )
   & lc_UC( *n, *nuc )
   & concatenate( 'T', *nuc, *f )
   & LISTS( age_of( *n, *a ), *f )
   & LISTS( registered( *n, *p, *y ), *f )
   & LISTS( completed( *n, *c ), *f )
   & LISTS( grade_of( *n, *c, *g ), *f )
   & LISTS( course_enrolled( *n, *c ), *f )
   & LISTS( name_id( *n ), DDBTDIR )
   & print( *n.' transcript saved' )
   & /.

 save_transcript <-
     print( 'transcript save failed' ).
/*
*/

 list_transcript <-
     get_id( *n )
   & print( 'Transcript of '.*n )
   & LISTS( age_of( *n, *a ) )
   & LISTS( registered( *n, *p, *y ) )
   & LISTS( completed( *n, *c ) )
   & LISTS( grade_of( *n, *c, *g ) )
   & LISTS( course_enrolled( *n, *c ) )
   & /.

/*
*/
```

```
create_transcript <-
   get_new_id( *n )
  & get_program_id( *p )
  & create_transcript_1( *n, *p )
  & /.

create_transcript <-
   print( 'Transcript not created' ).

create_transcript_1( *n, *p ) <-
   print( 'Enter admissions data (Type end to stop):' )
  & transaction
  & create_transcript_2( *n, *p ).

create_transcript_2( *n, *p ) <-
   derivable( eligible_for_admission( *n, *p ) )
  & assert( name_id( *n ) )
  & print( *n.' transcript created' ).

create_transcript_2( *n, *p ) <-
   print( *n.' not eligible for first year '.*p )
  & user( 'Augment admissions data?' )
  & print( 'Continue transcript creation...' )
  & RETRY( create_transcript_1( *n, *p ) ).
```

```
/*
 Department Data Base: data dictionary

 The DDB consists of three classes of information (see Chapter 2,
 §2.3.2): data dictionary (DD), integrity constraints (IC), and
 question-answering (QA).

 This is the data dictionary component (the topic definitions and
 synonym dictionary below can also be considered as data dictionary
 information).

*/

user_predicate( student_program_contribution, 3 ).
user_predicate( completed, 2 ).
user_predicate( program_enrolled, 2 ).
user_predicate( registered, 3 ).
user_predicate( elective, 2 ).
user_predicate( dept_program_prereq, 6 ).
user_predicate( matriculation_course, 1 ).
user_predicate( course_enrolled, 2 ).
user_predicate( field_of, 2 ).
user_predicate( year_of, 2 ).
user_predicate( recommended, 3 ).
user_predicate( faculty_of, 2 ).
user_predicate( course, 1 ).
user_predicate( program_of, 2 ).
user_predicate( program_req, 3 ).
user_predicate( stream_of, 2 ).
user_predicate( level, 1 ).
user_predicate( year, 1 ).
user_predicate( faculty_grad_req, 4 ).
user_predicate( program_prereq, 3 ).
user_predicate( faculty_elective, 3 ).
user_predicate( unit_value, 2 ).
user_predicate( faculty_course_req, 5 ).
user_predicate( course_equivalent, 2 ).
user_predicate( faculty_program_req, 5 ).
user_predicate( faculty_program_prereq, 5 ).
user_predicate( program_contribution, 3 ).
user_predicate( eligible_for_degree, 2 ).
user_predicate( eligible_for_course, 2 ).
user_predicate( eligible_for_admission, 2 ).
user_predicate( eligible_for_year, 2 ).
user_predicate( field, 1 ).
user_predicate( dept_of, 2 ).
user_predicate( faculty, 1 ).
user_predicate( course_no, 2 ).
user_predicate( program, 1 ).
user_predicate( grad_req, 2 ).
user_predicate( level_of, 2 ).
user_predicate( stream, 1 ).
user_predicate( course_req, 3 ).
user_predicate( head, 1 ).
user_predicate( dept, 1 ).
user_predicate( previous_year, 2 ).
user_predicate( faculty_member, 1 ).
user_predicate( course_prereq, 2 ).
user_predicate( anticipated_credit, 3 ).
user_predicate( student, 1 ).
user_predicate( grade_of, 3 ).
user_predicate( degree, 1 ).
user_predicate( passing_grade_of, 2 ).
user_predicate( age_of, 2 ).
user_predicate( name_id, 1 ).
```

```
/*
 Department Data Base: Domain specification

 The DDB consists of three classes of information (see Chapter 2,
 §2.3.2): data dictionary (DD), integrity constraints (IC), and
 question-answering (QA).

 The QA compontent includes the domain specifications,
 the bachelor's requirements, bachelor's prerequisites, and general
 question-answering knowledge. This section includes the domain
 specifications.

*/

program( BScMajorsCS ).
program( BScHonoursCS ).
program( BScHonoursCSPHYS ).
program( BScHonoursCSMATH ).

/*
*/

level( Bachelor ).
level( Master ).
level( Doctor ).

/*
*/

stream( Majors ).
stream( Honours ).

/*
*/

field( CS ).
field( CSMATH ).
field( CSPHYS ).

/*
*/

faculty( Science ).
faculty( AppliedScience ).
faculty( Arts ).
faculty( Commerce ).

/*
*/

dept( CS ).
dept( MATH ).
dept( EE ).
dept( ENGL ).
dept( PHYS ).
dept( CHEM ).
dept( PSYC ).
dept( GEOG ).
dept( ANTH ).
dept( ASIA ).
dept( CHIN ).
dept( CWRI ).
dept( ECON ).
dept( CLAS ).

/*
*/

year( first ).
year( second ).
year( third ).
year( fourth ).

/*
*/

course( CS448 ).
course( CS435 ).
course( CS430 ).
course( CS422 ).
course( CS420 ).
course( CS414 ).
course( CS413 ).
course( CS411 ).
course( CS410 ).
course( CS407 ).
course( CS406 ).
course( CS405 ).
course( CS404 ).
course( CS403 ).
course( CS402 ).
course( CS350 ).
course( CS321 ).
course( CS315 ).
course( CS313 ).
course( CS312 ).
course( CS311 ).
course( CS302 ).
course( CS251 ).
course( CS220 ).
course( CS215 ).
course( CS200 ).
course( CS118 ).
course( CS115 ).
course( CS101 ).
course( MATH100 ).
course( MATH101 ).
course( MATH120 ).
course( MATH121 ).
course( MATH205 ).
course( MATH221 ).
course( MATH300 ).
course( MATH305 ).
course( MATH306 ).
course( MATH307 ).
course( MATH315 ).
course( MATH316 ).
course( MATH318 ).
course( MATH320 ).
course( MATH340 ).
course( MATH344 ).
course( MATH345 ).
course( MATH400 ).
course( MATH405 ).
course( MATH407 ).
course( MATH426 ).
course( MATH413 ).
course( MATH480 ).
course( COMM356 ).
course( COMM410 ).
course( COMM411 ).
course( COMM450 ).
course( COMM459 ).
course( EE256 ).
course( EE358 ).
course( EE364 ).
course( PHYS110 ).
course( PHYS115 ).
course( PHYS120 ).
course( CHEM110 ).
course( CHEM120 ).
course( ENGL100 ).
course( GEOG101 ).
course( GEOG212 ).
course( GEOG213 ).
course( GEOG311 ).
course( GEOG312 ).
course( GEOG313 ).
course( GEOG316 ).
course( GEOG379 ).
course( GEOG410 ).
course( GEOG411 ).
course( GEOG412 ).
course( GEOG413 ).
course( GEOG414 ).
course( GEOG416 ).
course( GEOG447 ).
course( GEOG449 ).
course( GEOG500 ).
course( GEOG504 ).
course( GEOG505 ).
course( GEOG516 ).
course( GEOG521 ).
course( GEOG522 ).
course( GEOG525 ).
course( GEOG555 ).
course( GEOG560 ).
course( GEOG561 ).
course( PSYC260 ).
course( PSYC360 ).
course( PSYC366 ).
course( PSYC460 ).
course( PSYC463 ).
course( PSYC466 ).
course( PSYC467 ).
course( PSYC348 ).
course( PSYC448 ).
course( ANTH100 ).
course( ANTH200 ).
course( ANTH201 ).
course( ANTH202 ).
course( ANTH203 ).
course( ANTH213 ).
course( ASIA105 ).
course( ASIA115 ).
course( ASIA206 ).
course( CHIN100 ).
course( CLAS100 ).
course( CLAS210 ).
course( CWRI202 ).
course( ECON100 ).
course( FART125 ).
course( FART181 ).

/*
*/

matriculation_course( ALGEBRA11 ).
matriculation_course( ALGEBRA12 ).
matriculation_course( CHEM11 ).
matriculation_course( CHEM12 ).
matriculation_course( PHYS11 ).
matriculation_course( PHYS12 ).
matriculation_course( BIOL11 ).
matriculation_course( BIOL12 ).

/*
Department Data Base: Integrity constraints

Another of the three components of the DDB (Chapter 2, §2.3.2)
is the following set of integrity constraints.
```

```
*/

  course_enrolled( *s, *c ) <=
    course( *c )
   & ( registered( *s, *p, * ) | eligible_for_admission( *s, *p ) )
   & satisfied( *s,
               set( *x: course_prereq( *c, *x ) ) )
   & student_program_contribution( *s, *p, *c ).

/*
*/

 registered( *s, *p, *y ) <=
    satisfied( *s,
              set( *pr :
                  program_prereq( *p, *y, *pr ) ) )
   & local_ctx( completed( *s, *c ) <- course_enrolled( *s, *c ),
              derivable( satisfied( *s,
                                   set( *cr :
                                       course_req( *p, *y, *cr ) ) ) ) ).

/*  NOTE transition constraint
*/

 completed( *s, *c ) <=
    matriculation_course( *c )
   | course_enrolled( *s, *c )
   & grade_of( *s, *c, *g )
   & passing_grade_of( *c, *p )
   & GE( *g, *p )
   & DELAX( course_enrolled( *s, *c ) ).

/*
*/

 recommended( *p, *y, *c ) <=
    program_contribution( *p, *y, *c ).

/*
*/

 student( *s ) <=
    registered( *s, *, * ).

/*
*/

 degree( *d ) <=
    level_of( *d, *l )
   & faculty_of( *d, *fa )
   & field_of( *d, *fi ).
```

```
/*
*/

level_of( *x, *l ) <=
    ( program( *x ) | student( *x ) )
  & level( *l ).

level_of( *x, Bachelor ) <=
    stream_of( *x, Honours )
  | stream_of( *x, Majors ).

/*
*/

faculty_of( *x, *f ) <=
    ( dept( *x ) | student( *x ) | program( *x ) )
  & faculty( *f ).

/*
*/

dept_of( *x, *d ) <=
    ( course( *x ) | student( *x ) | faculty_member( *x ) )
  & dept( *d ).

/*
*/

field_of( *x, *f ) <=
    ( program( *x ) | student( *x ) | degree( *x ) )
  & field( *f ).

/*
*/

stream_of( *x, *s ) <=
    ( program( *x ) | student ( *x ) )
  & stream( *s ).

/*
*/

head( *h, *d ) <=
    faculty_member( *h )
  & dept( *d ).

/*
*/

grade_of( *s, *c, *g ) <=
    student( *s )
  & course( *c )
  & grade( *g ).
```

```
/*
*/

passing_grade_of( *c, *g ) <=
    course( *c )
  & grade( *g ).
```

```
/*
  Department Data Base: bachelor's requirements


  The DDB consists of three classes of information (see Chapter 2,
  §2.3.2): data dictionary (DD), integrity constraints (IC), and
  question-answering (QA).

  The QA compontent includes the domain specifications,
  the bachelor's requirements, bachelor's prerequisites, and general
  question-answering knowledge. This section includes the bachelor's
  requirements.

*/

 grad_req( *p, *req ) <-
    faculty_of( *p, *fac )
   & level_of( *p, *lev )
   & stream_of( *p, *str )
   & faculty_grad_req( *fac, *lev, *str, *req ).

 grad_req( *p, *req ) <-
    dept_of( *p, *dep )
   & level_of( *p, *lev )
   & stream_of( *p, *str )
   & field_of( *p, *fie )
   & dept_grad_req( *dep, *lev, *str, *fie, *req ).

/*
 faculty graduation requirements - Science
*/

faculty_grad_req( Science,
              Bachelor,
              Majors,
              lambda( *s,
                     completed( *s,
                              set( *c ; set( *s ) : course( *c )
                                 & ( unit_value( set( *s ), *v )
                                    & GE( *v, 600 ) ) ) ) ) ).

faculty_grad_req( Science,
              Bachelor,
              Honours,
              lambda( *s,
                     completed( *s,
                              set( *c ; set( *s ) : course( *c )
                                 & ( unit_value( set( *s ), *v )
                                    & GE( *v, 660 ) ) ) ) ) ).

faculty_grad_req( Science,
              Bachelor,
              *str,
                  lambda( *s,
                        completed( *s,
                                 set( *c ; set( *s ) :
                                     ( course( *c )
                                       & faculty_of( *c, Science ) )
                                   & ( unit_value( set( *s ), *v )
                                       & GE( *v, 360 ) ) ) ) ) ).

faculty_grad_req( Science,
                  Bachelor,
                  *str,
                  lambda( *s,
                        completed( *s,
                                 set( *c ; set( *s ) :
                                     ( course( *c )
                                       & faculty_of( *c, Arts ) )
                                   & ( unit_value( set( *s ), *v )
                                       & GE( *v, 90 ) ) ) ) ) ).

faculty_grad_req( Science,
                  Bachelor,
                  *str,
                  lambda( *s,
                        completed( *s,
                                 set( *c ; set( *s ) :
                                     ( course( *c )
                                       & course_no( *c, *n )
                                       & GE( *n, 300 ) )
                                   & ( unit_value( set( *s ), *v )
                                       & GE( *v, 210 ) ) ) ) ) ).

faculty_grad_req( Science,
                  Bachelor,
                  *str,
                  lambda( *s,
                        completed( *s,
                                 set( *c ; set( *s ):
                                     ( course( *c )
                                       & course_no( *c, *n )
                                       & GE( *n, 300 )
                                       & faculty_of( *c, Science ) )
                                   & ( unit_value( set( *c ), *v )
                                       & GE( *v, 150 ) ) ) ) ) ).
/*
 program requirements
*/

program_req( *p, *yea, *req ) <-
   faculty_of( *p, *fac )
  & level_of( *p, *lev )
  & stream_of( *p, *str )
  & faculty_program_req( *fac, *lev, *str, *yea, *req ).
```

```
program_req( *p, *yea, *req ) <-
   dept_of( *p, *dep )
  & level_of( *p, *lev )
  & stream_of( *p, *str )
  & field_of( *p, *fie )
  & dept_program_req( *dep, *lev, *str, *fie, *yea, *req ).

program_req( *p, *yea, *req ) <-
   course_req( *p, *yea, *req ).

/*
 faculty program requirements - Science
*/

faculty_program_req( Science,
                Bachelor,
                *str,
                first,
                lambda( *s,
                      completed( *s,
                              set( *c ; set( *s ) :
                                  course( *c )
                                 & ( unit_value( set( *s ), *v )
                                    & GE( *v, 90 ) ) ) ) ) ).

faculty_program_req( Science,
                Bachelor,
                *str,
                first,
                lambda( *s,
                      completed( *s,
                              set( *c ; set( *s ):
                                  course( *c )
                                 & ( unit_value( set( *s ), *v )
                                    & GE( *v, 60 )
                                    & extension( set( ( CHEM110
                                                     ! CHEM120 )
                                                    & MATH100
                                                    & MATH101
                                                    & ( PHYS110
                                                     ! PHYS115
                                                     ! PHYS120 ) ),
                                                 set( *x ) )
                                    & subset( set( *s ),
                                           set( *x ) ) ) ) ) ) ).

faculty_program_req( Science,
                Bachelor,
                *str,
                second,
                lambda( *s,
                      completed( *s,
                              set( *c ; set( *s ) :
                                course( *c )
                              & ( unit_value( set( *s ), *v )
                                 & GE( *v, 240 ) ) ) ) ) ).
faculty_program_req( Science,
                     Bachelor,
                     *str,
                     second,
                     lambda( *s,
                             completed( *s,
                                        set( *c ; set( *s ):
                                             course( *c )
                                             ( & extension( set( ( CHEM110
                                                                  ! CHEM120 )
                                                                 & MATH100
                                                                 & MATH101
                                                                 & ( PHYS110
                                                                  ! PHYS115
                                                                  ! PHYS120 ) ),
                                                            set( *x ) )
                                               & subset( set( *s ), set( *x ) )
                                               & unit_value( set( *s ), *v )
                                               & GE( *v, 90 ) ) ) ) ) ).

faculty_program_req( Science,
                     Bachelor,
                     *str,
                     second,
                     lambda( *s,
                             completed( *s,
                                        set( *c ; set( *s ) :
                                             course( *c )
                                             ( & extension( set( ( CHEM110
                                                                  ! CHEM120 )
                                                                 & MATH100
                                                                 & MATH101
                                                                 & ( PHYS110
                                                                  ! PHYS115
                                                                  ! PHYS120 ) ),
                                                            set( *x ) )
                                               & subset( set( *s ), set( *x ) )
                                               & unit_value( set( *s ), *v )
                                               & GE( *v, 70 ) ) ) ) ) ).


faculty_program_req( Science,
                     Bachelor,
                     *str,
                     third,
                     lambda( *s,
                             completed( *s,
                                        set( *c ; set( *s ) :
                                             course( *c )
                                           & ( unit_value( set( *c ), *v )
                                        & GE( *v, 390 ) ) ) ) ) ).

faculty_program_req( Science,
                     Bachelor,
                     *str,
                     third,
                     lambda( *s,
                             completed( *s,
                                        set( *c ; set( *s ) :
                                             ( course( *c )
                                               & faculty_of( *c, Science ) )
                                           & ( unit_value( set( *c ), *v )
                                               & GE( *v, 250 ) ) ) ) ) ).


faculty_program_req( Science,
                     Bachelor,
                     Honours,
                     second,
                     lambda( *s,
                             completed( *s,
                                        set( *c ; set( *s ) :
                                             course( *c )
                                           & ( unit_value( set( *c ), *v )
                                               & GE( *v, 300 ) ) ) ) ) ).

faculty_program_req( Science,
                     Bachelor,
                     Honours,
                     third,
                     lambda( *s,
                             completed( *s,
                                        set( *c ; set( *s ) :
                                             course( *c )
                                           & ( unit_value( set( *c ), *v )
                                               & GE( *v, 450 ) ) ) ) ) ).

faculty_program_req( Science,
                     Bachelor,
                     *str,
                     fourth,
                     lambda( *s,
                             completed( *s,
                                        set( *c ; set( *s ) :
                                             course( *c )
                                           & ( unit_value( set( *c ), *v )
                                               & GE( *v, 600 ) ) ) ) ) ).


/*
 course requirements
*/
```

```
course_req( *pro, *yea, *req ) <-
   faculty_of( *pro, *fac )
  & level_of( *pro, *lev )
  & stream_of( *pro, *str )
  & faculty_course_req( *fac, *lev, *str, *yea, *req ).

course_req( *pro, *yea, *req ) <-
   dept_of( *pro, *dep )
  & level_of( *pro, *lev )
  & stream_of( *pro, *str )
  & field_of( *pro, *fie )
  & dept_course_req( *dep, *lev, *str, *fie, *yea, *req ).

/*
 faculty course requirements - Science
*/

faculty_course_req( Science,
                    Bachelor,
                    *str,
                    second,
                    lambda( *s,
                            completed( *s, ENGL100 ) ) ).

/*
 course requirements - BScMajorsCS
*/

course_req( BScMajorsCS,
            first,
            lambda( *s,
                    completed( *s, CS115 )
                    ! ( completed( *s, CS118 )
                      & completed( *s,
                                   an( *c, elective( BScMajorsCS, *c )
                                         & unit_value( *c, 15 ) ) ) ) ) ).

course_req( BScMajorsCS,
            first,
            lambda( *s,
                    completed( *s, MATH100 )
                  & completed( *s, MATH101 )
                  ! ( completed( *s, MATH120 )
                    & completed( *s, MATH121 ) ) ) ).

course_req( BScMajorsCS,
            first,
            lambda( *s,
                    completed( *s, PHYS110 )
                  ! completed( *s, PHYS115 )
                  ! completed( *s, PHYS120 ) ) ).

course_req( BScMajorsCS,
            first,
            lambda( *s,
                  completed( *s, CHEM110 )
                ! completed( *s, CHEM120 ) ) ).

course_req( BScMajorsCS,
            first,
            lambda( *s,
                  completed( *s, ENGL100 ) ) ).

course_req( BScMajorsCS,
            second,
            lambda( *s,
                  completed( *s, CS215 ) ) ).

course_req( BScMajorsCS,
            second,
            lambda( *s,
                  completed( *s, CS220 ) ) ).

course_req( BScMajorsCS,
            second,
            lambda( *s,
                  completed( *s, MATH205 ) ) ).

course_req( BScMajorsCS,
            second,
            lambda( *s,
                  completed( *s, MATH221 ) ) ).

course_req( BScMajorsCS,
            second,
            lambda( *s,
                  completed( *s,
                          an( *c,
                             course( *c )
                           & dept_of( *c, MATH )
                           & unit_value( *c, 15 ) ) ) ) ).

course_req( BScMajorsCS,
            second,
            lambda( *s,
                  completed( *s,
                          set( *c ; set( *s ) :
                              elective( BScMajorsCS, *c )
                            & unit_value( set( *s ), 60 ) ) ) ) ).

course_req( BScMajorsCS,
            set( third & fourth ),
            lambda( *s,
                  completed( *s,
                          set( *c ; set( *x ) :
                              ( course( *c )
                                 & dept_of( *c, CS )
                                 & course_no( *c, *n )
                                 & GE( *n, 300 ) )
                                & unit_value( set( *s ), 60 ) ) ) ) ) ).

course_req( BScMajorsCS,
          set( third & fourth ),
          lambda( *s,
                completed( *s,
                         set( *c ; set( *x ) :
                             ( course( *c )
                               & dept_of( *c, CS )
                               & course_no( *c, *n )
                               & GE( *n, 400 ) )
                             & unit_value( set( *s ), 60 ) ) ) ) ) ).

course_req( BScMajorsCS,
          set( third & fourth ),
          lambda( *s,
                completed( *s,
                         set( *c ; set( *x ) :
                             ( course( *c )
                               & dept_of( *c, MATH )
                               & course_no( *c, *n )
                               & GE( *n, 300 ) )
                             & unit_value( set( *s ), 60 ) ) ) ) ) ).

course_req( BScMajorsCS,
          set( third & fourth ),
          lambda( *s,
                completed( *s,
                         set( *c ; set( *x ) :
                               elective( BScMajorsCS, *c )
                             & unit_value( set( *s ), 90 ) ) ) ) ).
```

```
/*
  Department Data Base: bachelor's prerequisites


  The DDB consists of three classes of information (see Chapter 2,
  §2.3.2): data dictionary (DD), integrity constraints (IC), and
  question-answering (QA).

  The QA compontent includes the domain specifications,
  the bachelor's requirements, bachelor's prerequisites, and general
  question-answering knowledge. This section includes the bachelor's
  prerequisites.

*/

 program_prereq( *p,
                first,
                lambda( *s,
                        age_of( *s, *a ) & GE( *a, 16 ) ) ).

 program_prereq( *p, *yea, *pre ) <-
    faculty_of( *p, *fac )
  & level_of( *p, *lev )
  & stream_of( *p, *str )
  & faculty_program_prereq( *fac, *lev, *str, *yea, *pre ).

 program_prereq( *p, *yea, *pre ) <-
    dept_of( *p, *dep )
  & level_of( *p, *lev )
  & stream_of( *p, *str )
  & field_of( *p, *fie )
  & dept_program_prereq( *dep, *lev, *str, *fie, *yea, *pre ).

/*
  faculty program prerequisites
*/

faculty_program_prereq( Science,
                       Bachelor,
                       *str,
                       first,
                       lambda( *s,
                               completed( *s, CHEM11 ) ) ).

faculty_program_prereq( Science,
                       Bachelor,
                       *str,
                       first,
                       lambda( *s,
                               completed( *s, ALGEBRA11 ) ) ).

faculty_program_prereq( Science,
                       Bachelor,
                    *str,
                    first,
                    lambda( *s,
                           completed( *s, ALGEBRA12 ) ) ).

faculty_program_prereq( Science,
                    Bachelor,
                    *str,
                    first,
                    lambda( *s,
                           completed( *s, PHYS11 ) ) ).

faculty_program_prereq( Science,
                    Bachelor,
                    *str,
                    first,
                    lambda( *s,
                           completed( *s,
                                     an( *c,
                                        topic_of( *c, Science )
                                       & ( course_no( *c, 11 )
                                         | course_no( *c, 12 ) ) ) )
                                       | (  permission( *s, *d, Science )
                                         & dean( *d, Science ) ) ) ) ).

faculty_program_prereq( Science,
                    Bachelor,
                    Honours,
                    second,
                    lambda( *s,
                           program_of( *s, *p )
                         & dept_of( *p, *d )
                         & head_of( *d, *h )
                         & permission( *s, *h, *p ) ) ).

faculty_program_prereq( Science,
                    Bachelor,
                    Honours,
                    third,
                    lambda( *s,
                           program_of( *s, *p )
                         & dept_of( *p, *d )
                         & head_of( *d, *h )
                         & permission( *s, *h, *p ) ) ).

faculty_program_prereq( Science,
                    Bachelor,
                    Honours,
                    fourth,
                    lambda( *s,
                           program_of( *s, *p )
                         & dept_of( *p, *d )
                         & head_of( *d, *h )
```

```
                        & permission( *s, *h, *p ) ) ).

/*
 department program prerequisites
*/

 dept_program_prereq( CS,
                      Bachelor,
                      Honours,
                      *fie
                      *yea
                      lambda( *s,
                            permission( *s, *h, Honours )
                          & head( *h, CS ) ) ).

/*
 course prerequisites - Computer Science
*/

course_prereq( CS448,
          lambda( *s,
               permission( *s, *h, CS448 )
             & head( *h, CS ) ) ).

course_prereq( CS435,
          lambda( *s,
               permission( *s, *h, CS435 )
             & head( *h, CS )
             | completed( *s, CS215 ) ) ).

course_prereq( CS430,
          lambda( *s,
               permission( *s, *h, CS430 )
             & head( *h, CS )
             | ( completed( *s, *c )
               & dept( *c, CS ) ) ) ).

course_prereq( CS422,
          lambda( *s,
               completed( *s, CS215 )
             & ( completed( *s, CS312 )
               | course_enroll( *s, CS312 ) ) ) ).

course_prereq( CS420,
          lambda( *s,
               completed( *s, CS215 )
             & completed( *s, CS220 ) ) ).

course_prereq( CS414,
          lambda( *s,
               completed( *s, CS115 )
             | completed( *s, CS118 )
             | ( permission( *s, *h, CS414 )
                  & head( *h, CS ) ) ) ).

course_prereq( CS413,
          lambda( *s,
                 completed( *s, CS313 )
               | completed( *s, EE358 ) ) ).

course_prereq( CS411,
          lambda( *s,
                 completed( *s, CS311 )
               & completed( *s, CS313 )
               & completed( *s, CS315 ) ) ).

course_prereq( CS410,
          lambda( *s,
                 completed( *s, CS313 )
               & completed( *s, CS315 ) ) ).

course_prereq( CS407,
          lambda( *s,
                 year_of( *s, fourth ) ) ).

course_prereq( CS406,
          lambda( *s,
                 completed( *s, *h )
               & course_equivalent( *h, CS115 )
               & completed( *s, *c )
               & course_equivalent( *c, MATH340 )
               & completed( *s, MATH221 ) ) ).

course_prereq( CS405,
          lambda( *s,
                 completed( *s, *h )
               & course_equivalent( *h, CS115 )
               & completed( *s, *c )
               & course_equivalent( *c, MATH205 ) ) ).

course_prereq( CS404,
          lambda( *s,
                 (  permission( *s, *h, CS404 )
                  & head( *h, CS ) )
               | completed( *s, CS315 ) ) ).

course_prereq( CS403,
          lambda( *s,
                 completed( *s, CS302 )
               & (  completed( *s, MATH300 )
                  | completed( *s, MATH315 )
                  | completed( *s, MATH320 ) ) ) ).

course_prereq( CS402,
          lambda( *s,
                 completed( *s, CS302 )
```

```
                  & (  completed( *s, MATH300 )
                    | completed( *s, MATH315 )
                    | completed( *s, MATH320 ) ) ) ).

course_prereq( CS350,
          lambda( *s,
                  completed( *s, CS251 ) ) ).

course_prereq( CS321,
          lambda( *s,
                  completed( *s, CS115 )
                & completed( *s, CS220 ) ) ).

course_prereq( CS315,
          lambda( *s,
                  completed( *s, CS215 ) ) ).

course_prereq( CS313,
          lambda( *s,
                  completed( *s, CS215 ) ) ).

course_prereq( CS312,
          lambda( *s,
                  completed( *s, CS215 ) ) ).

course_prereq( CS311,
          lambda( *s,
                  completed( *s, CS215 ) ) ).

course_prereq( CS302,
          lambda( *s,
                  completed( *s, *h )
                & course_equivalent( *h, CS115 )
                & completed( *s, MATH200 )
                & completed( *s, MATH221 ) ) ).

course_prereq( CS251,
          lambda( *s,
                  program_of( *s, *h )
                & faculty_of( *h, AppliedScience ) ) ).

course_prereq( CS220,
          lambda( *s,
                  (  completed( *s, CS115 )
                   | completed( *s, CS118 ) )
                & (  completed( *s, MATH101 )
                   | (  completed( *s, CS118 )
                      & course_enrolled( *s, MATH101 ) ) ) ) ).

course_prereq( CS200,
          lambda( *s,
                  ~eligible( *s, CS115 ) ) ).
```

```
course_prereq( CS118,
           lambda( *s,
                 ( completed( *s, *c )
                  | course_enroll( *s, *c ) )
                & course_equivalent( *c, MATH100 )
                & ( completed( *s, CS101 )
                  | ( completed( *s, *c )
                    & course( *c )
                    & subject_of( *c, Programming ) ) )
                & ~completed( *s, CS115 ) ) ).

course_prereq( CS115,
           lambda( *s,
                 ( completed( *s, *c )
                  | course_enroll( *s, *c ) )
                & course_equivalent( *c, MATH100 )
                & ~completed( *s, CS118 ) ) ).

course_prereq( CS101,
           lambda( *s,
                 ( completed( *s, *c )
                  | course_enrolled( *s, *c ) )
                & dept_of( *c, MATH )
                & units( *c, 15 )
                & ~completed( *s, CS115 ) ) ).

/*
 course prerequisites - Mathematics
*/

course_prereq( MATH100,
           lambda( *s,
                 completed( *s, MATH12 )
                | completed( *s, ALGEBRA12 ) ) ).

course_prereq( MATH101,
           lambda( *s,
                 completed( *s, MATH100 )
                | course_enrolled( *s, MATH100 )
                | completed( *s, MATH111 ) ) ).

course_prereq( MATH120,
           lambda( *s,
                 ( completed( *s, MATH12 )
                  | completed( *s, ALGEBRA12 )
                & permission( *s, *h, MATH120 )
                & head( *h, MATH ) ) ).

course_prereq( MATH121,
           lambda( *s,
                 ( completed( *s, MATH12 )
                  | completed( *s, ALGEBRA12 ) )
                & permission( *s, *h, MATH121 )
                & head( *h, MATH ) ) ).

course_prereq( MATH205,
          lambda( *s,
                completed( *s, MATH101 ) ) ).

course_prereq( MATH221,
          lambda( *s,
                completed( *s, MATH101 ) ) ).

course_prereq( MATH300,
          lambda( *s,
                completed( *s, MATH200 )
              & completed( *s, MATH221 )
              & (  course_enrolled( *s, MATH220 )
                | (  completed( *s, MATH301 )
                  | completed( *s, MATH316 ) ) ) ) ).

course_prereq( MATH315,
          lambda( *s,
                completed( *s, MATH200 )
              & completed( *s, MATH221 )
              & ~completed( *s, MATH165 ) ) ).

course_prereq( MATH320,
          lambda( *s,
                completed( *s, MATH200 )
              & completed( *s, MATH220 )
              & completed( *s, MATH221 )
              & grade_of( *s, MATH200, *h )
              & grade_of( *s, MATH220, *c )
              & grade_of( *s, MATH221, *w )
              & GE( *h, second )
              & GE( *c, second )
              & GE( *w, second ) ) ).

course_prereq( MATH340,
          lambda( *s,
                completed( *s, MATH221 ) ) ).

course_prereq( MATH413,
          lambda( *s,
                (  permission( *s, *h, MATH413 )
                 & head( *h, CS ) )
              | (  completed( *s, set( *c ; set( *x ) :
                                     dept_of( *c, MATH )
                                   & units( set( *c ), 120 ) ) ) ) ) ).

/*
 course prerequisites - Physics
*/

course_prereq( PHYS110,
            lambda( *s,
                  completed( *s, PHYS11 )
                & ( course_enrolled( *s, MATH100 )
                  | completed( *s, MATH100 ) )
                & ( course_enrolled( *s, MATH101 )
                  | completed( *s, MATH101 ) ) ) ).

course_prereq( PHYS115,
            lambda( *s,
                  completed( *s, PHYS11 )
                & completed( *s, PHYS12 )
                & ( course_enrolled( *s, MATH100 )
                  | completed( *s, MATH100 ) )
                & ( course_enrolled( *s, MATH101 )
                  | completed( *s, MATH101 ) ) ) ).

course_prereq( PHYS110,
            lambda( *s,
                  completed( *s, PHYS12 )
                & grade_of( *s, PHSY12, *g1 )
                & GE( *g1, 80 )
                & completed( *s, ALGEBRA12 )
                & grade_of( *s, ALGEBRA12, *g2 )
                & GE( *g2, 80 )
                & ( course_enrolled( *s, MATH100 )
                  | completed( *s, MATH100 ) )
                & ( course_enrolled( *s, MATH101 )
                  | completed( *s, MATH101 ) ) ) ).

/*
 course prerequisites - Chemistry
*/

course_prereq( CHEM110,
            lambda( *s,
                  completed( *s, CHEM11 )
                & completed( *s, CHEM12 )
                & ( course_enrolled( *s, MATH100 )
                  | completed( *s, MATH100 ) )
                & ( course_enrolled( *s, MATH101 )
                  | completed( *s, MATH101 ) )
                & ( course_enrolled( *s,
                              an( *c,
                                  course( *c )
                                & dept_of( *c, PHYS )
                                & course_no( *c, *n )
                                & nth_digit( 1, *n, 1 ) ) )
                  | completed( *s, an( *c,
                                 course( *c )
                               & dept_of( *c, PHYS )
                               & course_no( *c, *n )
                               & nth_digit( 1, *n, 1 ) ) ) ) ) ).
```

```
course_prereq( CHEM120,
        lambda( *s,
             completed( *s, CHEM11 )
            & completed( *s, CHEM12 )
            & completed( *s, PHYS11 )
            & ( course_enrolled( *s, MATH100 )
              | completed( *s, MATH100 ) )
            & ( course_enrolled( *s, MATH101 )
              | completed( *s, MATH101 ) )
            & ( course_enrolled( *s,
                         an( *c,
                            course( *c )
                          & dept_of( *c, PHYS )
                          & course_no( *c, *n )
                          & nth_digit( 1, *n, 1 ) ) )
              | completed( *s, an( *c,
                          course( *c )
                        & dept_of( *c, PHYS )
                        & course_no( *c, *n )
                        & nth_digit( 1, *n, 1 ) ) ) ) ) ).

/*
 course prerequisites - English
*/

course_prereq( ENGL100,
        lambda( *s,
             eligible_for_admission( *s, *p ) ) ).
```

```
/*
  Department Database: general QA knowledge


  The DDB consists of three classes of information (see Chapter 2,
  §2.3.2): data dictionary (DD), integrity constraints (IC), and
  question-answering (QA).

  The QA compontent includes the domain specifications,
  the bachelor's requirements, bachelor's prerequisites, and general
  question-answering knowledge. This section includes the general
  question-answering knolwedge.

*/

student( *s ) <-
   registered( *s, *, * ).

/*
*/

program_of( *s, *p ) <-
   registered( *s, *p, * ).

/*
*/

program_enrolled( *s, *p ) <-
   registered( *s, *p, * ).

/*
*/

year_of( *s, *y ) <-
   registered( *s, *, *y ).

/*
*/

previous_year( second, first ).
previous_year( third, second ).
previous_year( fourth, third ).

/*
*/

eligible_for_degree( *s, *d ) <-
   student( *s )
  & program_enrolled( *s, *d )
  & satisfied( *s,
           set( *gr : grad_req( *d, *gr ) ) ).

eligible_for_course( *s, *c ) <-
```

```
      course( *c )
    & (  registered( *s, *p, * )
      | eligible_for_admission( *s, *p ) )
    & satisfied( *s,
              set( *cr : course_prereq( *c, *cr ) ) )
    & student_program_contribution( *s, *p, *c ).

  eligible_for_admission( *s, *p ) <-
      program( *p )
    & satisfied( *s,
              set( *pr : program_prereq( *p, first, *pr ) ) ).

  eligible_for_year( *s, *y ) <-
      program_enrolled( *s, *p )
    & previous_year( *y, *py )
    & satisfied( *s,
              set( *pr : program_req( *p, *py, *pr ) ) )
    & satisfied( *s,
              set( *pq : program_prereq( *p, *y, *pq ) ) ).

/*
*/

  program_contribution( *p, *c ) <-
      program( *p )
    & course( *c )
    & course_req( *p, *, lambda( *, *cr ) )
    & extends( *cr, completed( *s, *c ) ).

/*
*/

  student_program_contribution( *s, *p, *c ) <-
      course_req( *p, *, lambda( *s, *r ) )
    & extends( *r, completed( *s, *c ) )
    & ~satisfied( *s, lambda( *s, *r ) ).

/*
*/

  anticipated_credit( *s, *c, *v ) <-
      student( *s )
    & course( *c )
    & unit_value( *c, *v )
    & individual_constant( *v ).

  anticipated_credit( *s, *c, *v ) <-
      student( *s )
    & course( *c )
    & unit_value( *c, *ia )
    & individual_aggregate( *ia )
    & user_select( *ia, *v ).
```

```
/*
*/

 elective( *p, *c ) <-
    faculty_of( *p, *fac )
  & faculty_elective( *fac, *fac1, *c ).

 elective( *p, *c ) <-
    dept_of( *p, *dep )
  & dept_elective( *dep, *dep1, *c ).

/*
*/

 unit_value( set( *c ), *v ) <-
    unit_value( *c, *v )
  | unit_value( *c, *v | * ).

 unit_value( set( *c & *rest ), *v ) <-
    unit_value( set( *c ), *v1 )
  & unit_value( set( *rest ), *v2 )
  & SUM( *v1, *v2, *v ).

/*
*/

 grade( *g ) <-
    INT( *g )
  & LE( *g, 100 )
  & GE( *g, 0 ).

/*
*/

 passing_grade_of( *c, 50 ) <-
   course( *c ).

/*
*/

faculty_of( BScMajorsCS, Science ).
faculty_of( BScHonoursCSMATH, Science ).
faculty_of( BScHonoursCSPHYS, Science ).
faculty_of( BScHonoursCS, Science ).

faculty_of( CS, Science ).
faculty_of( MATH, Science ).
faculty_of( EE, AppliedScience ).
faculty_of( ENGL, Arts ).
faculty_of( PHYS, Science ).
faculty_of( CHEM, Science ).
faculty_of( PSYC, Arts ).
faculty_of( GEOG, Arts ).
faculty_of( COMM, Commerce ).
faculty_of( ANTH, Arts ).
faculty_of( ASIA, Arts ).
faculty_of( CHIN, Arts ).
faculty_of( CWRI, Arts ).
faculty_of( ECON, Arts ).
faculty_of( CLAS, Arts ).

faculty_of( *c, *f ) <-
    course( *c )
  & dept_of( *c, *d )
  & faculty_of( *d, *f ).

/*
*/

stream_of( BScMajorsCS, Majors ).
stream_of( BScHonoursCSMATH, Honours ).
stream_of( BScHonoursCSPHYS, Honours ).
stream_of( BScHonoursCS, Honours ).

/*
*/

level_of( BScMajorsCS, Bachelor ).
level_of( BScHonoursCS, Bachelor ).
level_of( BScHonoursCSPHYS, Bachelor ).
level_of( BScHonoursCSMATH, Bachelor ).

/*
*/

dept_of( *c, *d ) <-
    course( *c )
  & letter_prefix( *d, *c )
  & dept( *d ).

/*
*/

field_of( BScMajorsCS, CS ).
field_of( BScHonoursCS, CS ).
field_of( BScHonoursCSPHYS, CSPHYS ).
field_of( BScHonoursCSMATH, CSMATH ).

/*
*/

topic_of( PHYS11, Science ).
topic_of( PHYS12, Science ).
topic_of( CHEM11, Science ).
topic_of( CHEM12, Science ).
topic_of( BIOL11, Science ).
topic_of( BIOL12, Science ).

/*
*/

previous_year( second, first ).
previous_year( third, second ).
previous_year( fourth, third ).

/*
*/

course_equivalent( MATH12, ALGEBRA12 ).
course_equivalent( CS115, CS118 ).
course_equivalent( CS118, CS115 ).
course_equivalent( FOR435, CS435 ).
course_equivalent( CS435, FOR435 ).
course_equivalent( EE478, CS414 ).
course_equivalent( CS414, EE478 ).
course_equivalent( EE476, CS413 ).
course_equivalent( CS413, EE476 ).

course_equivalent( *c, *c ) <-
   course( *c ).

/*
*/

course_no( *c, *n ) <-
   ( course( *c )
    | matriculation_course( *c ) )
 & digit_suffix( *n, *c ).

/*
*/

unit_value( CS448, 15 | 30 ).
unit_value( CS435, 15 ).
unit_value( CS430, 15 ).
unit_value( CS422, 30 ).
unit_value( CS420, 15 ).
unit_value( CS414, 15 ).
unit_value( CS413, 15 ).
unit_value( CS411, 30 ).
unit_value( CS410, 30 ).
unit_value( CS407, 15 ).
unit_value( CS406, 15 ).
unit_value( CS405, 15 ).
unit_value( CS404, 15 ).
unit_value( CS403, 15 ).
unit_value( CS402, 15 ).
unit_value( CS350, 10 ).
unit_value( CS321, 15 ).
unit_value( CS315, 30 ).
unit_value( CS313, 15 ).
unit_value( CS312, 15 ).
unit_value( CS311, 15 ).
unit_value( CS302, 30 ).
unit_value( CS251, 10 ).
unit_value( CS220, 15 ).
unit_value( CS215, 30 ).
unit_value( CS200, 15 ).
unit_value( CS118, 15 ).
unit_value( CS115, 30 ).
unit_value( CS101, 15 ).
unit_value( MATH100, 15 ).
unit_value( MATH101, 15 ).
unit_value( MATH120, 15 ).
unit_value( MATH121, 15 ).
unit_value( MATH205, 15 ).
unit_value( MATH221, 15 ).
unit_value( MATH300, 30 ).
unit_value( MATH305, 15 ).
unit_value( MATH306, 15 ).
unit_value( MATH307, 15 ).
unit_value( MATH315, 15 ).
unit_value( MATH316, 15 ).
unit_value( MATH318, 30 ).
unit_value( MATH320, 30 ).
unit_value( MATH340, 15 ).
unit_value( MATH344, 15 ).
unit_value( MATH345, 15 ).
unit_value( MATH400, 30 ).
unit_value( MATH405, 30 ).
unit_value( MATH407, 15 ).
unit_value( MATH413, 30 ).
unit_value( MATH426, 30 ).
unit_value( MATH480, 15 ).
unit_value( COMM356, 30 ).
unit_value( COMM410, 15 ).
unit_value( COMM411, 15 ).
unit_value( COMM450, 15 ).
unit_value( COMM459, 30 ).
unit_value( EE256, 15 ).
unit_value( EE358, 15 ).
unit_value( EE364, 15 ).
unit_value( PHYS110, 30 ).
unit_value( PHYS115, 30 ).
unit_value( PHYS120, 30 ).
unit_value( CHEM110, 30 ).
unit_value( CHEM120, 30 ).
unit_value( ENGL100, 15 ).
unit_value( GEOG101, 30 ).
unit_value( GEOG212, 15 ).
unit_value( GEOG213, 15 ).
unit_value( GEOG311, 15 ).
unit_value( GEOG312, 15 ).
unit_value( GEOG313, 15 ).
unit_value( GEOG316, 15 ).
unit_value( GEOG379, 15 ).
unit_value( GEOG410, 15 ).
unit_value( GEOG411, 15 ).
unit_value( GEOG412, 15 ).
unit_value( GEOG413, 15 ).
unit_value( GEOG416, 15 ).
unit_value( GEOG447, 15 ).
unit_value( GEOG449, 30 ).
unit_value( GEOG500, 15 ).
unit_value( GEOG504, 15 ).
unit_value( GEOG505, 15 ).
unit_value( GEOG516, 15 ).
unit_value( GEOG521, 15 ).
unit_value( GEOG522, 15 ).
unit_value( GEOG525, 15 ).
unit_value( GEOG560, 15 | 30 ).
unit_value( GEOG561, 15 ).
unit_value( PSYC260, 30 ).
unit_value( PSYC360, 30 ).
unit_value( PSYC366, 30 ).
unit_value( PSYC460, 30 ).
unit_value( PSYC463, 30 ).
unit_value( PSYC466, 30 ).
unit_value( PSYC467, 15 | 30 ).
unit_value( PSYC348, 15 | 20 | 25 | 30 ).
unit_value( PSYC448, 15 | 20 | 25 | 30 ).
unit_value( ANTH100, 30 ).
unit_value( ANTH200, 30 ).
unit_value( ANTH201, 30 ).
unit_value( ANTH202, 30 ).
unit_value( ANTH203, 15 ).
unit_value( ANTH213, 30 ).
unit_value( ASIA105, 30 ).
unit_value( ASIA115, 30 ).
unit_value( ASIA206, 30 ).
unit_value( CHIN100, 30 ).
unit_value( CLAS100, 30 ).
unit_value( CLAS210, 30 ).
unit_value( CWRI202, 30 ).
unit_value( ECON100, 30 ).
unit_value( FART125, 30 ).
unit_value( FART181, 30 ).

/*
*/

faculty_elective( Science, Science, GEOG101 ).
faculty_elective( Science, Science, GEOG212 ).
faculty_elective( Science, Science, GEOG213 ).
faculty_elective( Science, Science, GEOG311 ).
faculty_elective( Science, Science, GEOG312 ).
faculty_elective( Science, Science, GEOG313 ).
faculty_elective( Science, Science, GEOG316 ).
faculty_elective( Science, Science, GEOG379 ).
faculty_elective( Science, Science, GEOG410 ).
faculty_elective( Science, Science, GEOG411 ).
faculty_elective( Science, Science, GEOG412 ).
faculty_elective( Science, Science, GEOG413 ).
faculty_elective( Science, Science, GEOG414 ).
faculty_elective( Science, Science, GEOG416 ).
faculty_elective( Science, Science, GEOG447 ).
faculty_elective( Science, Science, GEOG449 ).
faculty_elective( Science, Science, GEOG500 ).
faculty_elective( Science, Science, GEOG504 ).
faculty_elective( Science, Science, GEOG505 ).
faculty_elective( Science, Science, GEOG516 ).
faculty_elective( Science, Science, GEOG521 ).
faculty_elective( Science, Science, GEOG522 ).
faculty_elective( Science, Science, GEOG525 ).
faculty_elective( Science, Science, GEOG555 ).
faculty_elective( Science, Science, GEOG560 ).
faculty_elective( Science, Science, GEOG561 ).
faculty_elective( Science, Science, GEOG561 ).
faculty_elective( Science, Science, PSYC348 ).
faculty_elective( Science, Science, PSYC448 ).
faculty_elective( Science, Science, COMM356 ).
faculty_elective( Science, Science, COMM410 ).
faculty_elective( Science, Science, COMM411 ).
faculty_elective( Science, Science, COMM450 ).
faculty_elective( Science, Science, COMM459 ).
faculty_elective( Science, Science, EE256 ).
faculty_elective( Science, Science, EE358 ).
faculty_elective( Science, Science, EE364 ).
faculty_elective( Science, Arts, ANTH100 ).
faculty_elective( Science, Arts, ANTH200 ).
faculty_elective( Science, Arts, ANTH201 ).
faculty_elective( Science, Arts, ANTH202 ).
faculty_elective( Science, Arts, ANTH203 ).
faculty_elective( Science, Arts, ANTH213 ).
faculty_elective( Science, Arts, ASIA105 ).
faculty_elective( Science, Arts, ASIA206 ).
faculty_elective( Science, Arts, CHIN100 ).
faculty_elective( Science, Arts, CLAS100 ).
faculty_elective( Science, Arts, CLAS210 ).
faculty_elective( Science, Arts, CWRI202 ).
faculty_elective( Science, Arts, ECON100 ).
faculty_elective( Science, Arts, FART125 ).
faculty_elective( Science, Arts, FART181 ).

faculty_elective( Science, Science, *c ) <-
   course( *c )
  & dept_of( *c, PSYC )
  & digit_suffix( *n, *c )
  & nth_digit( 2, *n, *d )
  & GE( *d, 6 ).

/*
*/

recommended( BScMajorsCS, set( third & fourth ), MATH300 ).
recommended( BScMajorsCS, set( third & fourth ), MATH305 ).
recommended( BScMajorsCS, set( third & fourth ), MATH306 ).
recommended( BScMajorsCS, set( third & fourth ), MATH307 ).
recommended( BScMajorsCS, set( third & fourth ), MATH315 ).
recommended( BScMajorsCS, set( third & fourth ), MATH316 ).
recommended( BScMajorsCS, set( third & fourth ), MATH318 ).
recommended( BScMajorsCS, set( third & fourth ), MATH340 ).
recommended( BScMajorsCS, set( third & fourth ), MATH344 ).
recommended( BScMajorsCS, set( third & fourth ), MATH345 ).
recommended( BScMajorsCS, set( third & fourth ), MATH400 ).
recommended( BScMajorsCS, set( third & fourth ), MATH405 ).
recommended( BScMajorsCS, set( third & fourth ), MATH407 ).
recommended( BScMajorsCS, set( third & fourth ), MATH426 ).
recommended( BScMajorsCS, set( third & fourth ), MATH480 ).
recommended( BScMajorsCS, set( third & fourth ), COMM410 ).
recommended( BScMajorsCS, set( third & fourth ), COMM450 ).
recommended( BScMajorsCS, set( third & fourth ), COMM459 ).
recommended( BScMajorsCS, set( third & fourth ), COMM256 ).
recommended( BScMajorsCS, set( third & fourth ), EE256 ).
recommended( BScMajorsCS, set( third & fourth ), EE358 ).
recommended( BScMajorsCS, set( third & fourth ), EE364 ).

recommended( BScMajorsCS,
        set( third & fourth ),
        set( *c ; set( *s ) :
            ( course( *c )
             & element( *c, set( CS302
                              & CS402
                              & CS403
                              & CS404
                              & CS406 ) ) )
          & ( cardinality( set( *s ), *n )
             & GE( *n, 2 ) ) ) ).
```

```
/*
  Department Data Base: user predicate topic classification

  The DDB consists of three classes of information (see Chapter 2,
  §2.3.2): data dictionary (DD), integrity constraints (IC), and
  question-answering (QA).

  This section can be considered a portion of the data dictionary,
  as it provides information about the DDB.

*/

topic( advising ).
topic( registration ).
topic( courses ).
topic( standing ).
topic( admission ).
topic( promotion ).
topic( grades ).
topic( graduation ).

/*
*/

subtopic( advising, registration ).
subtopic( advising, courses ).
subtopic( advising, standing ).
subtopic( registration, admission ).
subtopic( registration, promotion ).
subtopic( standing, grades ).

/*
*/

topic_category( admission, program_prereq ).
topic_category( admission, faculty_program_prereq ).
topic_category( admission, dept_program_prereq ).
topic_category( promotion, grad_req ).
topic_category( promotion, faculty_grad_req ).
topic_category( promotion, dept_grad_req ).
topic_category( promotion, registered ).
topic_category( promotion, eligible_for_year ).
topic_category( promotion, eligible_for_degree ).
topic_category( courses, program_req ).
topic_category( courses, faculty_program_req ).
topic_category( courses, dept_program_req ).
topic_category( courses, program_contribution ).
topic_category( courses, student_program_contribution ).
topic_category( courses, eligible_for_course ).
topic_category( courses, course_prereq ).
topic_category( courses, course_equivalent ).
topic_category( courses, unit_value ).
topic_category( courses, elective ).
topic_category( courses, faculty_elective ).
topic_category( grades, completed ).
topic_category( grades, grade_of ).
topic_category( grades, passing_grade ).
topic_category( standing, registered ).
topic_category( standing, eligible_for_year ).
topic_category( standing, eligible_for_degree ).
topic_category( standing, completed ).
topic_category( standing, grade_of ).
topic_category( standing, course_enrolled ).
topic_category( standing, year_of ).
topic_category( standing, faculty_of ).
topic_category( standing, dept_of ).
topic_category( standing, level_of ).
topic_category( standing, field_of ).
topic_category( standing, stream_of ).
topic_category( registration, registered ).
topic_category( registration, eligible_for_admission ).
topic_category( registration, eligible_for_year ).
topic_category( graduation, eligible_for_degree ).

/*
*/

topic_category( *t, *c ) <-
   subtopic( *t, *st )
  & topic_category( *st, *c ).

/*
*/

topic_equivalent( admission, admissions ).
topic_equivalent( admission, admitting ).
topic_equivalent( promotion, promotions ).
topic_equivalent( promotion, promoting ).
topic_equivalent( courses, course ).
topic_equivalent( courses, classes ).
topic_equivalent( courses, class ).
topic_equivalent( grades, grade ).
topic_equivalent( grades, marks ).
topic_equivalent( grades, scores ).
topic_equivalent( registration, registering ).
topic_equivalent( registration, register ).
topic_equivalent( registration, registrations ).
topic_equivalent( registration, enrolling ).
topic_equivalent( registration, enrolment ).
topic_equivalent( advising, advice ).
topic_equivalent( advising, counsellor ).
topic_equivalent( advising, counselling ).
topic_equivalent( advising, counsellors ).
topic_equivalent( advising, counsel ).
topic_equivalent( advising, help ).
```