Logic Group
Report Logic-92-1

First Draft January 1992
Current Version June 1992

Knowledge Interchange Format
Version 3.0
Reference Manual

by

Michael R. Genesereth
Richard E. Fikes

in collaboration with

Daniel Bobrow
Ronald Brachman
Thomas Gruber
Patrick Hayes
Reed Letsinger
Vladimir Lifschitz
Robert MacGregor
John McCarthy
Peter Norvig
Ramesh Patil
Len Schubert

This manual is the "living document" of the Interlingua Working Group of the DARPA Knowledge Sharing Effort. As such, it represents work in progress toward a proposal for a standard knowledge interchange format.

Computer Science Department
Stanford University
Stanford, California 94305

Abstract: *Knowledge Interchange Format* (KIF) is a computer-oriented language for the interchange of knowledge among disparate programs. It has declarative semantics (i.e. the meaning of expressions in the representation can be understood without appeal to an interpreter for manipulating those expressions); it is logically comprehensive (i.e. it provides for the expression of arbitrary sentences in the first-order predicate calculus); it provides for the representation of knowledge about the representation of knowledge; it provides for the representation of nonmonotonic reasoning rules; and it provides for the definition of objects, functions, and relations.

Table of Contents

Chapter 1
Introduction

Knowledge Interchange Format (KIF) is a formal language for the interchange of knowledge among disparate computer programs (written by different programmers, at different times, in different languages, and so forth).

KIF is *not* intended as a primary language for interaction with human users (though it can be used for this purpose). Different programs can interact with their users in whatever forms are most appropriate to their applications (for example frames, graphs, charts, tables, diagrams, natural language, and so forth).

KIF is also *not* intended to be an internal representation for knowledge *within* computer programs or within closely related sets of programs (though it can be used for this purpose as well). Typically, when a program reads a knowledge base in KIF, it converts the data into its own internal form (specialized pointer structures, arrays, etc.). All computation is done using these internal forms. When the program needs to communicate with another program, it maps its internal data structures into KIF.

The purpose of KIF is roughly analogous to that of Postscript. Postscript is commonly used by text and graphics formatting programs in communicating information about documents to printers. Although it is not as efficient as a specialized representation for documents and not as perspicuous as a specialized wysiwyg display, Postscript is a programmer-readable representation that facilitates the independent development of formatting programs and printers. While KIF is not as efficient as a specialized representation for knowledge nor as perspicuous as a specialized display (when printed in its list form), it too is a programmer-readable language and thereby facilitates the independent development of knowledge-manipulation programs.

The definition of KIF is highly detailed. Some of these details are essential; others are arbitrary. The following general features are essential in the definition of KIF.

1. The language has declarative semantics. It is possible to understand the meaning of expressions in the language without appeal to an interpreter for manipulating those expressions. In this way, KIF differs from other languages that are based on specific interpreters, such as Emycin and Prolog.
2. The language is logically comprehensive – it provides for the expression of arbitrary sentences in predicate calculus. In this way, it differs from relational database languages (many of which are confined to ground atomic sentences) and Prolog-like languages (that are confined to Horn clauses).
3. The language provides for the representation of knowledge about the representation of knowledge. This allows us to make all knowledge representation decisions explicit and permits us to introduce new knowledge representation constructs without changing the language.

In addition to these hard criteria, KIF is designed to maximize in a joint fashion the following somewhat softer measures as well (to the extent possible while satisfying the preceding criteria).

1. Translatability. A central operational requirement for KIF is that it enable practical means of translating declarative knowledge bases to and from typical knowledge representation languages.
2. Readability. Although KIF is not intended primarily as a language for interaction with humans, human readability facilitates its use in describing representation language semantics, its use as a publication language for example knowledge bases, its use in assisting humans with knowledge base translation problems, etc.
3. Useability as a representation language. Although KIF is not intended for use within programs as a representation or communication language, it *can* be used for that purpose if so desired.

This document supplies full technical details of KIF. Chapter 2 presents the formal syntax of the language. Chapter 3 discusses conceptualizations of the world. Chapter 4 defines the semantics of the language. Chapter 5 deals with lists; chapter 6, with sets; and chapter 7, with functions and relations. Chapter 8 describes how metaknowledge is encoded. Chapter 9 describes the formalization of monotonic and nonmonotonic rules of inference. Chapter 10 discusses definitions.

Chapter 2
Syntax

Like many computer-oriented languages, KIF has two varieties. In *linear* KIF, all expressions are strings of ASCII characters and, as such, are suitable for storage on serial devices (such as magnetic disks) and for transmission on serial media (such as phone lines). In *structured* KIF, the legal "expressions" of the language are structured objects. Structured KIF is of special use in communication between programs operating in the same address space.

Fortunately, there is a simple correspondence between the two varieties of KIF. For every character string, there is exactly one corresponding list structure; and, for every list structure, there is exactly one corresponding character string (once all unnecessary white space is eliminated).

In what follows, we first define the mapping between the linear and structured forms of the language; and, thereafter, we deal exclusively with the structured form.

§2.1 Linear KIF

The alphabet of linear KIF consists of the 128 characters in the ASCII character set. Some of these characters have standard print representations; others do not. The characters with standard print representations (93 of the 128) are shown below.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 ( ) [ ] { } < >
= + - * / \ & ^ ~ ' ` " _ @ # $ % : ; , . ! ?
```

KIF originated in a Lisp application and inherits its syntax from Lisp. The relationship between linear KIF and structured KIF is most easily specified by appeal to the Common Lisp reader [Steele]. In particular, a string of ascii characters forms a legal expression in linear KIF if and only if (1) it is acceptable to the Common Lisp reader (as defined in Steele's book) and (2) the structure produced by the Common Lisp reader is a legal expression of structured KIF (as defined in the next section).

§2.2 Structured KIF

In structured KIF, the notion of *word* is taken as primitive. An *expression* is either a word or a finite sequence of expressions. In our treatment here, we use enclosing parentheses to bound the items in a composite expression.

```
<word> ::= a primitive syntactic object

<expression> ::= <word> | (<expression>*)
```

The set of all words is divided into the categories listed below. This categorization is disjoint and exhaustive. Every word is a member of one and only one category. (The categories defined here are used again in the grammatical rules of subsequent tables.)

<indvar> ::= *a word beginning with the character* ?

<seqvar> ::= *a word beginning with the character* @

<termop> ::= listof | setof | quote | if | cond |
the | setofall | kappa | lambda

<sentop> ::= = | /= | not | and | or | => | <= | <=> | forall | exists

<ruleop> ::= =>> | <<= | consis

<defop> ::= defobject | defunction | defrelation | := | :=> | :&

<objconst> ::= *a word denoting an object*

<funconst> ::= *a word denoting a function*

<relconst> ::= *a word denoting a relation*

<logconst> ::= *a word denoting a truth value*

From these fundamental categories, we can build up more complex categories, viz. variables, operators, and constants.

<variable> ::= <indvar> | <seqvar>

<operator> ::= <termop> | <sentop> | <ruleop> | <defop>

<constant> ::= <objconst> | <funconst> | <relconst> | <logconst>

A *variable* is a word in which the first character is ? or @. A variable that begins with ? is called an *individual variable*. A variable that begins with an @ is called a *sequence variable*. Individual variables are used in quantifying over individual objects. Sequence variables are used in quantifying over sequences of objects.

Operators are used in forming complex expressions of various sorts. There are four types of operators in KIF – *term operators*, *sentence operators*, *rule operators*, and *definition operators*. Term operators are used in forming complex terms. Sentence operators are used in forming complex sentences. Rule operators are using in forming rules. Definition operators are used in forming definitions.

A *constant* is any word that is neither a variable nor an operator. There are four categories of constants in KIF – object constants, function constants, relation constants, and logical constants. *Object constants* are used to denote individual objects. *Function constants* denote functions on those objects. *Relation constants* denote relations. *Logical constants* express conditions about the world and are either true or false.

Some constants are *basic* in that their type and meaning are fixed in the definition of KIF. All other constants are *non-basic* in that the language user gets to choose the type and the meaning. All numbers, characters, and strings are basic constants in KIF; the remaining basic constants are described in the remaining chapters of this document.

KIF is unusual among logical languages in that there is no way of determining the category of a non-basic constant (i.e. whether it is an object, function, relation, or logical constant) from its inherent properties (i.e. its spelling). The user selects the category of every non-basic constant for himself. The user need not declare that choice explicitly. However, the category of a constant determines how it can be used in forming expressions, and its category *can* be determined from this use. Consequently, once a constant is used in a particular way, its category becomes fixed.

There are four special types of expressions in the language – *terms*, *sentences*, *rules*, and *definitions*. Terms are used to denote objects in the world being described; sentences are used to express facts about the world; rules are used to express legal steps of inference; and definitions are used to define constants; and forms are either sentences, rules, or definitions.

The set of legal terms in KIF is defined below. There are ten types of terms – individual variables, object constants, function constants, relation constants, functional terms, list terms, set terms, quotations, logical terms, and quantified terms. Individual variables, object constants, function constants, and relation constants were discussed earlier.

```
<term> ::= <indvar> | <objconst> | <funconst> | <relconst>|
           <funterm> | <listterm> | <setterm> |
           <quoterm> | <logterm> | <quanterm>

<listterm> ::= (listof <term>* [<seqvar>])

<setterm> ::= (setof <term>* [<seqvar>])

<funterm> ::= (<funconst> <term>* [<seqvar>])

<quoterm> ::= (quote <expression>)

<logterm> ::= (if <sentence> <term> [<term>])|
              (cond (<sentence> <term>) ... (<sentence> <term>))

<quanterm> ::= (the <term> <sentence>)|
               (setofall <term> <sentence>)|
               (kappa (<indvar>* [<seqvar>]) <sentence>*)|
               (lambda (<indvar>* [<seqvar>]s) <term>)
```

A *functional term* consists of a function constant and an arbitrary number of *argument* terms, terminated by an optional sequence variable. Note that there is no syntactic restriction on the number of argument terms – the same function constant can be applied to different numbers of arguments; arity restrictions in KIF are treated semantically.

A *list term* consists of the `listof` operator and a finite list of terms, terminated by an optional sequence variable.

A *set term* consists of the `setof` operator and a finite list of terms, terminated by an optional sequence variable.

Quotations involve the `quote` operator and an arbitrary list expression. The embedded expression can be an arbitrary list structure; it need *not* be a legal expression in KIF. Remember that the Lisp reader converts strings of the form `'`σ into `(quote` σ`)`.

Logical terms involve the `if` and `cond` operators. The `if` form allows for the testing of a single condition only, whereas the `cond` form allows for the testing of a sequence of conditions.

Quantified terms involve the operators `the`, `setofall`, `kappa`, and `lambda`. A *designator* consists of the `the` operator, a term, and a sentence. A *set-forming term* consist of the `setof` operator, a term, and a sentence. A *relation-forming term* consists of `kappa`, a list of variables, and a sentence. A *function-forming term* consists of `lambda`, a list of variables, and a term. Strictly speaking, we do not need `kappa` and `lambda` – both can be defined in terms of `setof`; they are included in KIF for the sake of convenience.

The following BNF defines the set of legal sentences in KIF. There are six types of sentences. We have already mentioned logical constants.

```
<sentence> ::= <logconst>|<equation>|<inequality>|
               <relsent>|<logsent>|<quantsent>

<equation> ::= (= <term> <term>)

<inequality> ::= (/= <term> <term>)

<relsent> ::= (<relconst> <term>* [<seqvar>])|
              (<funconst> <term>* <term>)

<logsent> ::= (not <sentence>)|
              (and <sentence>*)|
              (or <sentence>*)|
              (=> <sentence>* <sentence>)|
              (<= <sentence> <sentence>*)|
              (<=> <sentence> <sentence>)

<quantsent> ::= (forall <indvar> <sentence>)|
                (forall (<indvar>*) <sentence>)|
                (exists <indvar> <sentence>)|
                (exists (<indvar>*) <sentence>)
```

An *equation* consists of the `=` operator and two terms.

An *inequality* consist of the `/=` operator and two terms.

A *relational sentence* consists of a relation constant and an arbitrary number of *argument* terms, terminated by an optional sequence variable. As with functional terms, there is no syntactic restriction on the number of argument terms in a relation sentence – the same relation constant can be applied to any finite number of arguments.

The syntax of *logical sentences* depends on the logical operator involved. A sentence involving the `not` operator is called a *negation*. A sentence involving the `and` operator is called a *conjunction*, and the arguments are called *conjuncts*. A sentence involving the `or` operator is called a *disjunction*, and the arguments are called *disjuncts*. A sentence involving the `=>` operator is called an *implication*; all of its arguments but the last are called *antecedents*; and the last argument is called the *consequent*. A sentence involving the `<=` operator is called a *reverse implication*; its first argument is called the *consequent*; and the remaining arguments are called the *antecedents*. A sentence involving the `<=>` operator is called an *equivalence*.

There are two types of *quantified sentences* – a *universally quantified sentence* is signalled by the use of the `forall` operator, and an *existentially quantified sentence* is signalled by the use of the `exists` operator.

The following BNF defines the set of legal KIF rules.

```
<rule> ::= (=>> <premise>* <sentence>) |
           (<<= <sentence> <premise>*)

<premise> ::= <sentence> | (consis <sentence>)
```

The last argument in a forward rule is called the *consequent* of the rule. Analogously, the first argument in a reverse rule is called the *consequent*. The premises that are sentences are its *prerequisites*, and the premises that have the form (consis ϕ) are its *justifications*.

The following BNF defines the set of legal KIF definitions.

```
<definition> ::= <complete> | <partial>

<complete> ::=
 (defobject <objconst> := <term>) |
 (deffunction <funconst> (<indvar>* [<seqvar>]) := <term>) |
 (defrelation <relconst> (<indvar>* [<seqvar>]) := <sentence>)

<partial> :: <conservative> | <unrestricted>

<conservative> ::=
 (defobject <objconst> [:conservative-axiom <sentence>]) |
 (deffunction <funconst> [:conservative-axiom <sentence>]) |
 (defrelation <relconst> [:conservative-axiom <sentence>])|
 (defrelation <relconst> (<indvar>* [<seqvar>])
   :=><sentence> [:conservative-axiom <sentence>])

<unrestricted> ::=
 (defobject <objconst> <sentence>*) |
 (deffunction <funconst> <sentence>*) |
 (defrelation <relconst> <sentence>*) |
 (defrelation <relconst> (<indvar>* [<seqvar>])
   :=> <sentence> [:axiom <sentence>])
```

Definitions are used to make category declarations and specify *defining axioms* for constants (e.g. "A triangle is a polygon with 3 sides."). KIF definitions can be *complete* in that they specify an expression that defines the concept completely, or they can be *partial* in that they constrain the concept without necessarily giving a complete equivalence. Partial definitions can be either *conservative* or *unrestricted*. Conservative definitions are restricted in that their addition to a knowledge base does not result in the logical entailment of any additional sentences not containing the constant being defined.

Object constants are defined using the `defobject` operator by specifying (1) a term that is equivalent to the constant or (2) a sentence that provides a partial description of the object denoted by the constant. Function constants are defined using the `deffunction` operator by specifying (1) a term that is equivalent to the function applied to a given set of arguments or (2) a sentence that provides a partial description of the function denoted by the constant. Relation constants are defined using the `defrelation` operator by specifying (1) necessary and sufficient conditions for the relation to hold, (2) necessary conditions for the relation to hold, or (3) arbitrary sentences describing the relation.

A *form* in KIF is either a sentence, a rule, or a definition.

```
<form> ::= <sentence> | <definition> | <rule>
```

A *knowledge base* is a finite set of forms. It is important to keep in mind that a knowledge base is a *set* of sentences, not a *sequence*; the order of forms within the knowledge base is unimportant.

Chapter 3
Conceptualization

The formalization of knowledge in KIF, as in any declarative representation, requires a *conceptualization* of the world in terms of objects, functions, and relations.

§3.1 Objects

A *universe of discourse* is the set of all objects presumed or hypothesized to exist in the world. The notion of *object* used here is quite broad. Objects can be concrete (e.g. a specific carbon atom, Confucius, the Sun) or abstract (e.g. the number 2, the set of all integers, the concept of justice). Objects can be primitive or composite (e.g. a circuit that consists of many subcircuits). Objects can even be fictional (e.g. a unicorn, Sherlock Holmes).

Different users of a declarative representation language, like KIF, are likely to have different universes of discourse. KIF is *conceptually promiscuous* in that it does *not* require every user to share the same universe of discourse. On the other hand, KIF is *conceptually grounded* in that every universe of discourse *is* required to include certain *basic* objects.

The following basic objects must occur in every universe of discourse.

- Words. Yes, the words of KIF are themselves objects in the universe of discourse, along with the things they denote.
- All complex numbers.
- All finite lists of objects in the universe of discourse.
- All sets of objects in the universe of discourse.
- $\perp$ (pronounced "bottom") – a distinguished object that occurs as the value of various functions when applied to arguments for which the functions make no sense.

Remember, however, that to these basic elements, the user can add whatever *non-basic* objects seem useful.

§3.2 Functions and Relations

A function is one kind of interrelationship among objects. For every finite sequence of objects (called the *arguments*), a *function* associates a unique object (called the *value*). More formally, a function is defined as a set of finite lists of objects, one for each combination of possible arguments. In each list, the initial elements are the arguments, and the final element is the value. For example, the 1+ function contains the list $\langle 2, 3\rangle$, indicating that integer successor of 2 is 3.

A relation is another kind of interrelationship among objects in the universe of discourse. More formally, a *relation* is an arbitrary set of finite lists of objects (of possibly varying lengths). Each list is a selection of objects that jointly satisfy the relation. For example, the $<$ relation on numbers contains the list $\langle 2, 3\rangle$, indicating that 2 is less than 3.

Note that both functions and relations are defined as sets of lists. In fact, every function is a relation. However, not every relation is a function. In a function, there cannot be two lists that disagree on only the last element. This would be tantamount to the function having two values for one combination of arguments. By contrast, in a relation, there can be any number of lists that agree on all but the last element. For example, the list $\langle 2, 3\rangle$ is a member of the 1+ function, and there is no other list of length 2 with 2 as its first argument, i.e. there is only one successor for 2. By contrast, the $<$ relation contains the lists $\langle 2, 3\rangle$, $\langle 2, 4\rangle$, and so forth, indicating that 2 is less than 3, 4, and so forth.

Many mathematicians require that functions and relations have fixed arity, i.e they require that all of the lists comprising a function or relation have the same length. The definitions here allow for functions and relations with variable arity, i.e. it is perfectly acceptable for a function or a relation to contain lists of different lengths. For example, the $+$ function contains the lists $\langle 1, 1, 2\rangle$ and $\langle 1, 1, 1, 3\rangle$, reflecting the fact that the sum of 1 and 1 is 2 and the fact that the sum of 1 and 1 and 1 is 3. Similarly, the relation $<$ contains the lists $\langle 1, 2\rangle$ and $\langle 1, 2, 3\rangle$, reflecting the fact that 1 is less than 2 and the fact that 1 is less than 2 and 2 is less than 3. This flexibility is not essential, but it is extremely convenient and poses no significant theoretical problems.

Chapter 4
Semantics

Intuitively, the semantics of KIF is very simple. Unfortunately, the formal details are quite complex. Consequently, we proceed gradually in our presentation. In this chapter, we introduce the basic notions underlying the semantics of KIF (in particular, the notions of interpretation, variable assignment, semantic value, truth value, and various types of entailment).

The basis for KIF semantics is a correlation between the terms and sentences of the language and a conceptualization of the world. Every term denotes an object in the universe of discourse associated with the conceptualization, and every sentence is either true or false.

When we encode knowledge in KIF, we select constants on the basis of our understanding of their meanings. In some cases (e.g. the basic constants of the language), these meanings are fixed in the definition of the language. In other cases (i.e. the non-basic constants), the meanings can vary from one user to another.

Given exact meanings for the constants of the language (whether they are the meanings in the definition of the language or our own concoctions), the semantics of KIF tells us the meaning of its complex expressions. We can unambiguously determine the referent of any term, and we can unambiguously determine the truth or falsity of any sentence.

Unfortunately, few of us have complete knowledge about the world. In keeping with traditional logical semantics, this is equivalent to not knowing the exact referent for every constant in the language. In such situations, we write sentences that reflect all of the meanings consistent with whatever knowledge we have. In such situations, the semantics of the language cannot pick out exact meanings for all expressions in the language, but it does place constraints on the meanings of complex expressions.

And, of course, the meanings we ascribe to non-basic constants may differ from those ascribed by others. However, we can convey our meanings to others by writing sentences to constrain those meanings in accordance with our usage. By writing more and more sentences, the set of possible referents for our constants is decreased.

In the remainder of this section, we provide precise definitions for the ideas just introduced. We start off with a definition for the *interpretation* of constants, and we introduce the related notion of *variable assignment*. We then show how these concepts are used in defining the *semantic value* of terms and the *truth value* of sentences. Finally, we introduce several approaches to *entailment*, which eliminates the dependence of meaning on the interpretation of non-basic constants.

§4.1 Interpretation

An *interpretation* is a function i that associates the constants of KIF with the elements of a conceptualization. In order to be an interpretation, a function must satisfy the following two properties.

First, the function must map constants into concepts of the appropriate type. It must map object constants into objects in the universe of discourse. It must map function constants into functions on the universe of discourse. It must map relation constants into relations on the universe of discourse. Notice that we allow for functions and relations of variable, finite arity. The function must map logical constants into one of the boolean values *true* or *false* (which may or may not be members of the universe of discourse).

1. If σ is an object constant, then $i(\sigma) \in O$.
2. If σ is a function constant, then $i(\sigma) : O^* \longrightarrow O$.
3. If σ is a relation constant, then $i(\sigma) \subseteq O^*$.
3. If σ is a logical constant, then $i(\sigma) \in \{true, false\}$.

Second, i must "satisfy" the conditions and axioms given in this chapter and the remaining chapters of this document. As a start, this includes the following conditions.

Every interpretation must map every numerical constant σ into the corresponding number n (assuming base 10).

$$i(\sigma) = n$$

Every interpretation must map the object constant `bottom` into $\perp$.

$$i(\texttt{bottom}) = \perp$$

Every interpretation must map the logical constant `true` into *true* and the logical constant `false` into *false*.

$$i(\texttt{true}) = true$$

$$i(\texttt{false}) = false$$

Note that, even with these restrictions, KIF is only a "partially interpreted" language. Although the interpretations of some constants (the basic constants) are constrained in the definition of the language, the meanings of other constants (the non-basic constants) are left open (i.e. left to the imaginations of the language users).

§4.2 Variable Assignment

A *variable assignment* is a function that (1) maps individual variables $\mathcal{V}$ into objects in a universe of discourse O and (2) maps sequence variables $\mathcal{W}$ into finite sequences of objects.

$$v : \mathcal{V} \longrightarrow O$$

$$v : \mathcal{W} \longrightarrow O^*$$

The notion of a variable assignment is important in defining the meaning of quantified terms and sentences and is discussed further below.

§4.3 Semantic Value

Given an interpretation and a variable assignment, we can assign a *semantic value* to every term in the language. We formalize this as a function s_{iv} from the set $\mathcal{T}$ of terms into the set O of objects in the universe of discourse.

$$s_{iv} : \mathcal{T} \longrightarrow O$$

If an expression is an individual variable ν, the semantic value is the object assigned to that variable by the given variable assignment.

$$s_{iv}(\nu) = v(\nu)$$

The semantic value of an object constant σ is the object assigned to that constant by the given interpretation.

$$s_{iv}(\sigma) = i(\sigma)$$

The semantic value of a function constant π is the set of tuples in the universe of discourse corresponding to the function denoted by π. Here, we use the operator `lambda` to denote this function. A full description of the semantics of expressions involving `lambda` is given later.

$$s_{iv}(\pi) = s_{iv}(\texttt{(lambda (@l) (}\pi\texttt{ @l))})$$

The semantic value of a relation constant ρ is the set of tuples in the universe of discourse corresponding to the relation denoted by ρ. Here, we use the operator `kappa` to denote this relation. A full description of the semantics of expressions involving `kappa` is given later.

$$s_{iv}(\rho) = s_{iv}(\texttt{(kappa (@l) (}\rho\texttt{ @l))})$$

In most cases, the semantic value of a function or relation constant is the same as its interpretation. However, in order to avoid paradoxes, it must in some cases be different. See the chapter on sets for a fuller discussion of this subject.

The semantic value of a functional term without a terminating sequence variable is obtained by applying the function denoted by the function constant in the term to the objects denoted by the arguments.

$$s_{iv}(\texttt{(}\pi\ \ \tau_1 \ldots \tau_n\texttt{)}) = i(\pi)[s_{iv}(\tau_1), ..., s_{iv}(\tau_n)]$$

If a functional term has a terminating sequence variable, the semantic value is obtained by applying the function to the sequence of arguments formed from the values of the terms that precede the sequence variable and the values in the sequence denoted by the sequence variable. (The vertical bar | here means that the objects in the sequence following the bar are appended to the sequence of elements before the bar.)

$$s_{iv}(\texttt{(}\pi\ \ \tau_1 \ldots \tau_n\ \ \omega\texttt{)}) = i(\pi)[s_{iv}(\tau_1), ..., s_{iv}(\tau_n)|s_{iv}(\omega)]$$

A term that begins with *listof* refers to the sequence of objects denoted by the arguments in the term. There is no restriction on the objects in the sequence.

$$s_{iv}(\texttt{(listof } \tau_1 \ \ldots \ \tau_k\texttt{)}) = \langle s_{iv}(\tau_1), ..., s_{iv}(\tau_k)\rangle$$

If a term that begins with `listof` ends with a sequence variable, the value of the term as a whole is the sequence consisting of the objects denoted by the terms prior to the sequence variable together with the objects in the sequence denoted by the sequence variable.

$$s_{iv}(\texttt{(listof } \tau_1 \ \ldots \ \tau_k \ \omega\texttt{)}) = \langle s_{iv}(\tau_1), ..., s_{iv}(\tau_k) | s_{iv}(\omega)\rangle$$

A term that begins with *setof* refers to the set of "bounded" objects denoted by the arguments in the term. The concept of boundedness is discussed further in the chapter on sets.

$$s_{iv}(\texttt{(setof } \tau_1 \ \ldots \ \tau_k\texttt{)}) = \{s_{iv}(\tau_1), ..., s_{iv}(\tau_k)\}$$

If a term that begins with `setof` ends with a sequence variable, the value of the term as a whole is the set consisting of the bounded objects denoted by the terms prior to the sequence variable together with the bounded objects in the sequence denoted by the sequence variable.

$$s_{iv}(\texttt{(setof } \tau_1 \ \ldots \ \tau_k \ \omega\texttt{)}) = \{s_{iv}(\tau_1), ..., s_{iv}(\tau_k)\} \cup \{x | x = s_{iv}(\omega)_n\}$$

A quotation denotes the expression contained as argument of the `quote` operator. Remember that the universe of discourse for every interpretation must contain all list expressions and that the argument to `quote` can be any list expression, whether or not it is a legal expression in KIF.

$$s_{iv}(\texttt{(quote } \epsilon\texttt{)}) = \epsilon$$

Note that any KIF expression (other than a word) is a sequence of KIF expressions. Thus, there are two ways it can be denoted – with `quote` and with `listof`. This means we have the following equivalence.

$$s_{iv}(\texttt{(quote (}\epsilon_1 \ \ldots \ \epsilon_n\texttt{))}) = s_{iv}(\texttt{(listof (quote } \epsilon_1\texttt{)} \ \ldots \ \texttt{(quote } \epsilon_n\texttt{))})$$

The semantic value of a simple conditional term depends on the truth value of the embedded sentence (see next section). If the truth value of the embedded sentence is *true*, then the semantic value of the term as a whole is the semantic value of the first term; otherwise, it is the semantic value of the second term (if there is one).

$$s_{iv}(\texttt{(if } \phi \ \tau_1 \ \tau_2\texttt{)}) = \begin{cases} s_{iv}(\tau_1) & t_{iv}(\phi) = true \\ s_{iv}(\tau_2) & \text{otherwise} \end{cases}$$

If a simple conditional has only one embedded term and the truth value of the embedded sentence is *true*, then the semantic value of the term is the semantic value of the embedded term. Otherwise, the value is $\perp$.

$$s_{iv}(\texttt{(if}\ \phi\ \tau_1\texttt{)}) = \begin{cases} s_{iv}(\tau_1) & t_{iv}(\phi) = true \\ \perp & \text{otherwise} \end{cases}$$

The semantic value of a complex conditional is the semantic value of the *first* term for which the truth value of the corresponding sentence is *true*. If none of the sentences are true, the semantic value is $\perp$.

$$s_{iv}(\texttt{(cond}\ \texttt{(}\phi_1\ \tau_1\texttt{)}\ \ldots\ \texttt{(}\phi_n\ \tau_n\texttt{)))} = \begin{cases} s_{iv}(\tau_1) & t_{iv}(\phi_1) = true \\ \cdots & \cdots \\ s_{iv}(\tau_2) & t_{iv}(\phi_n) = true \\ \perp & \text{otherwise} \end{cases}$$

The semantic value of a quantified term with an interpretation i and variable assignment v is determined by the semantic value of the embedded term or the truth value of the embedded sentence under the same interpretation but with various new versions of the variable assignment. We say that a variable assignment v' is a *version* of variable assignment v with respect to variables ν_1, ..., ν_n if and only if v' agrees with v on all variables except for ν_1, ..., ν_n. The assignments for ν_1, ..., ν_n can be the same as those in v or can be completely different.

The referent of a designator with term τ as first argument and sentence ϕ can be one of two things. Consider all versions v' of v with respect to the free variables in τ. If there is at least one version v' that makes ϕ true and the semantic value of τ is the same in every v' that makes ϕ true, then the semantic value of the designator as a whole is that value. If there is more than one such value, the semantic value is $\perp$.

$$s_{iv}(\texttt{(the}\ \tau\ \phi\texttt{)}) = \begin{cases} s_{iv'}(\tau) & t_{iv'}(\phi) = true \text{ and} \\ & s_{iv''}(\tau) = s_{iv'}(\tau) \text{ for all } v''\ t_{iv''}(\phi) = true \\ \perp & \text{otherwise} \end{cases}$$

A set-forming term with the term τ as first argument and the sentence ϕ as second argument denotes the set of objects in the universe of discourse with the following properties. (1) The object must be the semantic value of τ for some version v' of v that makes ϕ true. (2) The object must be *bounded*. A term is *bounded* if and only if it satisfies the interpretation of the `bounded` relation. See the chapter on sets for the axioms characterizing this relation.

$$s_{iv}(\texttt{(setofall}\ \tau\ \phi\texttt{)}) = \{s_{iv'}(\tau) | t_{iv'}(\phi) = true, s_{iv'}(\tau) \in i(\texttt{bounded})\}$$

A function-forming term denotes the set of tuples of bounded objects corresponding to the function that maps every tuple of objects matching the first argument of the term into the semantic value of the second argument.

$$s_{iv}(\texttt{(lambda}\ \texttt{(}\nu_1 \ldots \nu_n\texttt{)}\ \tau\texttt{)}) = s_{iv}(\texttt{(setofall}\ \texttt{(listof}\ \nu_1 \ldots \nu_n\ \nu\texttt{)}\ \texttt{(=}\ \tau\ \nu\texttt{))})$$

If the argument list of the function-forming term terminates in a sequence variable, the semantic value of the term is the union of the infinite series of sets of tuples corresponding to (1) the same term in which all occurrences of the sequence variable are dropped, (2) the same term in which all occurrences of the sequence variable are replaced by a single individual variable, (3) the same term in which all occurrences of the sequence variable are replaced by two individual variables, etc.

A relation-forming term denotes the set of all tuples of bounded objects that satisfy the embedded sentence.

$$s_{iv}(\texttt{(kappa (}\nu_1 \ldots \nu_n\ \texttt{[}\sigma\texttt{])}\ \phi\texttt{)}) = s_{iv}(\texttt{(setofall (listof}\ \nu_1 \ldots \nu_n\ \texttt{[}\sigma\texttt{])}\ \phi\texttt{)})$$

§4.4 Truth Value

In a manner similar to that for terms, we define the *truth value* for sentences in the language as a function t_{iv} that maps sentences $\mathcal{S}$ into the truth values $true$ or $false$.

$$t_{iv} : \mathcal{S} \longrightarrow \{true, false\}$$

The truth value of a logical constant is the truth value assigned by the corresponding interpretation.

$$t_{iv}(\lambda) = i(\lambda)$$

An equation is true if and only if the terms in the equation refer to the same object in the universe of discourse.

$$t_{iv}(\texttt{(=}\ \tau_1\ \ \tau_2\texttt{)}) = \begin{cases} true & s_{iv}(\tau_1) = s_{iv}(\tau_2) \\ false & \text{otherwise} \end{cases}$$

An inequality is true if and only if the terms in the equation refer to distinct objects in the universe of discourse.

$$t_{iv}(\texttt{(/=}\ \tau_1\ \ \tau_2\texttt{)}) = \begin{cases} false & s_{iv}(\tau_1) = s_{iv}(\tau_2) \\ true & \text{otherwise} \end{cases}$$

The truth value of a simple relational sentence without a terminating sequence variable is $true$ if and only if the relation denoted by the relation constant in the sentence is true of the objects denoted by the arguments. Equivalently, viewing a relation as a set of tuples, we say that the truth value of a relational sentence is $true$ if and only if the tuple of objects formed from the values of the arguments is a member of the set of tuples denoted by the relation constant.

$$t_{iv}(\texttt{(}\rho\ \tau_1\ \ldots\ \tau_n\texttt{)}) = \begin{cases} true & \langle s_{iv}(\tau_1), ..., s_{iv}(\tau_n)\rangle \in i(\rho) \\ false & \text{otherwise} \end{cases}$$

If a relational sentence terminates in a sequence variable, the sentence is true if and only if the relation contains the tuple consisting of the values of the terms that precede the sequence variable together with the objects in the sequence denoted by the variable. Remember that the vertical bar | means that the objects in the sequence following the bar are appended to the sequence of elements before the bar.

$$t_{iv}(\texttt{(}\rho\ \tau_1\ \ldots\ \tau_n\ \omega\texttt{)}) = \begin{cases} true & \langle s_{iv}(\tau_1), ..., s_{iv}(\tau_n) | s_{iv}(\omega)\rangle \in i(\rho) \\ false & \text{otherwise} \end{cases}$$

The truth value of a negation is *true* if and only if the truth value of the negated sentence is *false*.

$$t_{iv}(\texttt{(not}\ \phi\texttt{)}) = \begin{cases} true & t_{iv}(\phi) = false \\ false & \text{otherwise} \end{cases}$$

The truth value of a conjunction is *true* if and only if the truth value of every conjunct is *true*.

$$t_{iv}(\texttt{(and}\ \phi_1\ \ldots\ \phi_n\texttt{)}) = \begin{cases} true & t_{iv}(\phi_j) = true \text{ for all } j\ 1 \leq j \leq n \\ false & \text{otherwise} \end{cases}$$

The truth value of a disjunction is *true* if and only if the truth value of at least one of the disjuncts is *true*.

$$t_{iv}(\texttt{(or}\ \phi_1\ \ldots\ \phi_n\texttt{)}) = \begin{cases} true & t_{iv}(\phi_j) = true \text{ for some } j\ 1 \leq j \leq n \\ false & \text{otherwise} \end{cases}$$

If the truth value of every antecedent in an implication is *true*, then the the truth value of the implication as a whole is *true* if and only if the truth value of the consequent is *true*. If any of the antecedents is *false*, then the implication as a whole is *true*, regardless of the truth value of the consequent.

$$t_{iv}(\texttt{(=>}\ \phi_1\ \ldots\ \phi_n\ \phi\texttt{)}) = \begin{cases} true & \text{for some } j\ t_{iv}(\phi_j) = false \text{ or } t_{iv}(\phi) = true \\ false & \text{otherwise} \end{cases}$$

A reverse implication is just an implication with the consequent and antecedents reversed.

$$t_{iv}(\texttt{(<=}\ \phi\ \phi_1\ \ldots\ \phi_n\texttt{)}) = \begin{cases} true & t_{iv}(\phi) = true \text{ or for some } j\ t_{iv}(\phi_j) = false \\ false & \text{otherwise} \end{cases}$$

The truth value of an equivalence is *true* if and only if the embedded sentences have the same truth value.

$$t_{iv}(\texttt{(<=>}\ \phi_1\ \phi_2\texttt{)}) = \begin{cases} true & t_{iv}(\phi_1) = t_{iv}(\phi_2) \\ false & \text{otherwise} \end{cases}$$

Given an interpretation i and variable assignment v, the truth value of an existentially quantified sentence is *true* if and only if the truth value of the second argument is *true* for *some* version v' of variable assignment v with respect to the variables in the first argument.

$$t_{iv}(\texttt{(exists (}\nu_1 \ \ldots \ \ \nu_k \ \omega\texttt{)} \ \phi\texttt{))} = \begin{cases} true & \exists v' \, t_{iv'}(\phi) = true \\ false & \text{otherwise} \end{cases}$$

Given an interpretation i and variable assignment v, the truth value of a universally quantified sentence is *true* if and only if the truth value of the second argument of the sentence is *true* for *every* version v' of v with respect to variables in the first argument.

$$t_{iv}(\texttt{(forall (}\nu_1 \ \ldots \ \ \nu_k \ \omega\texttt{)} \ \phi\texttt{))} = \begin{cases} true & \forall v' \, t_{iv'}(\phi) = true \\ false & \text{otherwise} \end{cases}$$

§4.5 Logical Entailment

The definition of truth value relies on both an interpretation for the constants of KIF and an assignment for its variables. In encoding knowledge, we often have in mind a specific interpretation for the constants in our language, but we want our variables to range over the universe of discourse (either existentially or universally). In order to provide a notion of semantics that is independent of the assignment of variables, we turn to the notion of satisfaction.

An interpretation i *logically satisfies* a sentence ϕ if and only if the truth value of the sentence is *true* for all variable assignments. Whenever this is the case, we say that i is a *model* of ϕ. Extending this notion to sets of sentences, we say that an interpretation is a model of a set of sentences if and only if it is a model of every sentence in the set of sentences.

Obviously, a variable assignment has no effect on the truth value of a sentence without free variables (i.e. a ground sentence or one in which all variables are bound). Consequently, if an interpretation satisfies such a sentence for one variable assignment, it satisfies it for every variable assignment.

The occurrence of free variables in a sentence means that the sentence is true for all assignments of the variables. For example, the sentence `(p $x)` means that the relation denoted by `p` is true for all objects in the universe of discourse. In other words, the meaning of a sentence with free variables is the same as the meaning of a universally quantified sentence in which all of the free variables are boundby the universal quantifier. In KIF, we use this fact to sanction the dropping of prefix universal quantifiers that do not occur inside the scope of existential quantifiers. In other words, we are permitted to write `(=> (apple $x) (red $x))` in place of the more cumbersome `(forall ($x) (=> (apple $x) (red $x)))`.

Unfortunately, the notion of satisfaction is disturbing in that it is relative to an interpretation. As a result, different individuals and different programs with different interpretations may disagree on the truth of a sentence.

It is true that, as we add more sentences to a knowledge base, the set of models generally decreases. The goal of knowledge encoding is to write enough sentences so that unwanted interpretations are eliminated. Unfortunately, this is not always possible. In the light of this fact, how are we to interpret the expressions in such situations? The answer is to generalize over interpretations as earlier we generalized over variable assignments.

If Δ is a set of sentences, we say that Δ *logically entails* a sentence ϕ if and only every model of Δ is also a model of ϕ.

With this notion, we can rephrase the goal of knowledge representation as follows. It is to encode enough sentences so that every conclusion we desire is logically entailed by our set of sentences. It is a sad fact that this is not always possible, but it is the ideal toward which we strive.

§4.6 Indexical Entailment

In the definition of logical entailment, all interpretations are taken into account; there is no constraint. In certain situations, it is desirable to restrict the possible interpretations to those in which certain constants are assigned values having to do with the set of sentences itself. In this case, the constants are said to be *indexical*. An interpretation then is *indexical* if and only if it assigns these indexical constants correctly.

In KIF, there is a single indexical constant, viz. the object constant `knowedge-base`. An indexical interpretation of a knowledge base Δ is one in which this constant is assigned Δ as value. This one indexical makes it possible for the user to write sentences that depend on the knowledge base within which the sentences are contained.

Finally, we say that a set of sentences *indexically entails* a conclusion if and only if every indexical interpretation and variable assignment that satisfies the set of sentences also satisfies the conclusion.

§4.7 Nonmonotonic Entailment

Recall that the truth value of a sentence is defined relative to an interpretation i and a variable assignment v. To define the nonmonotonic value of a premise in a rule, we need to select, instead of a single interpretation i, a *set* of interpretations – the interpretations that are considered "possible". In the following definition, I is a set of interpretations which all have the same universe of discourse O, and v is a variable assignment with this universe. We consider prerequisites and justifications separately.

The nonmonotonic value of a prerequisite is *true* if and only if it is true at every "possible" intepretation.

$$n_{Iv}(\phi) = \begin{cases} true & \forall i \in I \; t_{iv}(\phi) = true \\ false & \text{otherwise} \end{cases}$$

The nonmonotonic value of a justification is *true* if and only if its argument is true for at least one "possible" intepretation.

$$n_{Iv}(\texttt{(consis } \phi\texttt{)}) = \begin{cases} true & \exists i \in I \; t_{iv}(\phi) = true \\ false & \text{otherwise} \end{cases}$$

Let Δ be a knowledge base with rules. We define when a set I of interpretations is "a set of possible worlds" for Δ, by means of the following fixpoint construction. Consider a universe of discourse O; by a *world* we understand an interpretation with the universe O. Let I be the set of all worlds that satisfy the sentences in Δ. Consider a maximal set I' of worlds such that, for each rule $\delta \in \Delta$ and each variable assignment v with the universe

O, the following condition holds. If the nonmonotonic value of every prerequisite of δ for I' and v is *true*, and the nonmonotonic value of every justification of δ for I and v is true, then the nonmonotonic value of the consequent of δ for I' and v is *true*. (This I' always exists.) If I' is maximal, then we say that I' is *a set of possible worlds* for Δ. Typically, a knowledge base with rules has many sets of possible worlds; it is clear, for instance, that any two interpretations with different universes cannot belong to the same set of possible worlds.

An interpretation i is a *nonmonotonic model* of Δ if it belongs to some set of possible worlds for Δ. We say that a nonmonotonic knowledge base Δ *nonmonotonically entails* a sentence ϕ if and only every nonmonotonic model of Δ is also a model of ϕ.

Note that the definition of a model for nonmonotonic knowledge bases is "nonlocal" – we cannot check whether an interpretation i is a model by looking at each rule in isolation. This feature of the definition is responsible for the nonmonotonic character in this notion of entailment.

§4.8 Definitions

The definitional operators in KIF allow us to state sentences that are true "by definition" in a way that distinguishes them from sentences that express contingent properties of the world. Definitions have no truth values in the sense described above. They are so because we say that they are so.

On the other hand, definitions have content – sentences that allow us to derive other sentences as conclusions. In KIF, every definition has a corresponding set of sentences, called the *content* of the definition. In general, there are three parts to this content.

First of all, there is information about the category of the constant in the definition. If the constant is a function constant or a relation constant, there is also information about its arity.

Second, there is the *defining axiom* associated with the definition (see below).

Finally, there is a sentence stating that the defining axiom associated with the definition is indeed a defining axiom for the associated concept (named by the constant σ used in the definition). The following sentence expresses this fact. Note the use of quotes to capture the fact that this is a relationship between a constant and a sentence.

(defining-axiom 'σ 'ϕ)

The rules for determining the defining axioms for a definition are somewhat complicated and are described fully in the chapter on definitions. The following is a brief outline, sufficient to enable the reader to understand the use of definitional constructs in the intervening chapters.

The `defobject` operator is used to define objects. The two simplest forms are shown below, together with their defining axioms. In the first case, the defining axiom is the equation involving the object constant in the definition with the defining term. In the second case, the defining axiom is the conjunction of the constituent sentences.

Definition	Defining Axiom
`(defobject` σ `:=` τ`)`	`(=` σ τ`)`
`(defobject` σ ϕ_1 ... ϕ_n`)`	`(and` ϕ_1 ... ϕ_n`)`

The `deffunction` operator is used to define functions. Again, the two simplest forms are shown below, together with their defining axioms. In the first case, the defining axiom is the equation involving (1) the term formed from the function constant in the definition and the variables in its argument list and (2) the defining term. In the second case, as with object definitions, the defining axiom is the conjunction of the constituent sentences.

Definition	Defining Axiom
`(deffunction` π `(`ν_1 ...ν_n`) :=` τ`)`	`(=` π `(lambda (`ν_1 ...ν_n`)` τ`))`
`(deffunction` π ϕ_1 ... ϕ_n`)`	`(and` ϕ_1 ... ϕ_n`)`

The `defrelation` operator is used to define relations. The two simplest forms are shown below, together with their defining axioms. In the first case, the defining axiom is the equivalence relating (1) the relational sentence formed from the relation constant in the definition and the variables in its argument list and (2) the defining sentence. In the second case, as with object and function definitions, the defining axiom is the conjunction of the constituent sentences.

Definition	Defining Axiom
`(defrelation` ρ `(`ν_1 ...ν_n`) :=` ϕ`)`	`(=` ρ `(kappa (`ν_1 ...ν_n`)` ϕ`))`
`(defrelation` ρ ϕ_1 ... ϕ_n`)`	`(and` ϕ_1 ... ϕ_n`)`

For most purposes, a definition can be viewed as shorthand for the sentences in the content of the definition.

Chapter 5
Numbers

KIF includes the following standard vocabulary for describing properties of numbers. *A formal axiomatization of numbers and of the associated functions and relations is being developed for inclusion in later versions of this manual. Common Lisp is being used as a guide in that development to determine both the types of numbers and the number-related functions and relations to include in the language. The informal descriptions below are provided to indicate the anticipated vocabulary.*

§5.1 Functions on Numbers

`*` - If τ_1, ..., τ_n denote numbers, then the term `(*` $\tau_1 \ldots \tau_n$`)` denotes the product of those numbers.

`+` - If τ_1, ..., τ_n are numerical constants, then the term `(+` $\tau_1 ... \tau_n$`)` denotes the sum τ of the numbers corresponding to those constants.

`-` - If τ and τ_1, ..., τ_n denote numbers, then the term `(-` τ $\tau_1 ... \tau_n$`)` denotes the difference between the number denoted by τ and the numbers denoted by τ_1 through τ_n. An exception occurs when $n = 0$, in which case the term denotes the negation of the number denoted by τ.

`/` - If τ_1, ..., τ_n are numbers, then the term `(/` $\tau_1 ... \tau_n$`)` denotes the result τ obtained by dividing the number denoted by τ_1 by the numbers denoted by τ_2 through τ_n. An exception occurs when $n = 1$, in which case the term denotes the reciprocal τ of the number denoted by τ_1.

`1+` - The term `(1+` τ`)` denotes the sum of the object denoted by τ and 1.

```
(deffunction 1+ (?x) :=  (+ ?x 1))
```
(5.1)

`1-` - The term `(1-` τ`)` denotes the difference of the object denoted by τ and 1.

```
(deffunction 1- (?x) := (- ?x 1))
```
(5.2)

`abs` - The term `(abs` τ`)` denotes the absolute value of the object denoted by τ.

```
(deffunction abs (?x) := (if (>= ?x 0) ?x (- ?x)))
```
(5.3)

`acos` - If τ denotes a number, then the term `(acos` τ`)` denotes the arc cosine of that number (in radians).

`acosh` - The term `(acosh` τ`)` denotes the arc cosine of the object denoted by τ (in radians).

`ash` - The term `(ash` τ_1 τ_2`)` denotes the result of arithmetically shifting the object denoted by τ_1 by the number of bits denoted by τ_2 (left or right shifting depending on the sign of τ_2).

`asin` - The term `(asin` τ`)` denotes the arc sine of the object denoted by τ (in radians).

`asinh` - The term `(asinh` τ`)` denotes the hyperbolic arc sine of the object denoted by τ (in radians).

`atan` - The term `(atan` τ`)` denotes the arc tangent of the object denoted by τ (in radians).

`atanh` - The term `(atanh` τ`)` denotes the hyperbolic arc tangent of the object denoted by τ (in radians).

`boole` - The term `(boole` τ τ_1 τ_2`)` denotes the result of applying the operation denoted by τ to the objects denoted by τ_1 and τ_2.

`ceiling` - If τ denotes a real number, then the term `(ceiling` τ`)` denotes the smallest integer greater than or equal to the number denoted by τ.

`cis` - The term `(cis` τ`)` denotes the complex number denoted by $cos(\tau) + isin(\tau)$. The argument is any non-complex number of radians.

`conjugate` - If τ denotes a complex number, then the term `(conjugate` τ`)` denotes the complex conjugate of the number denoted by τ.

```
(deffunction conjugate (?c) :=
  (complex-number (realpart ?c) (- (imagpart ?c))))
```
(5.4)

`cos` - The term `(cos` τ`)` denotes the cosine of the object denoted by τ (in radians).

`cosh` - The term `(cosh` τ`)` denotes the hyperbolic cosine of the object denoted by τ (in radians).

`decode-float` - The term `(decode-float` τ`)` denotes the mantissa of the object denoted by τ.

`denominator` - The term `(denominator` τ`)` denotes the denominator of the canonical reduced form of the object denoted by τ.

`exp` - The term `(exp` τ`)` denotes e raised to the power the object denoted by τ.

```
(deffunction exp (?x) :=  (expt e ?x))
```
(5.5)

`expt` - The term `(expt` τ_1 τ_2`)` denotes the object denoted by τ_1 raised to the power the object denoted by τ_2.

`fceiling` - The term `(fceiling` τ`)` denotes the smallest integer (as a floating point number) greater than the object denoted by τ.

`ffloor` - The term `(ffloor` τ`)` denotes the largest integer (as a floating point number) less than the object denoted by τ.

`float` - The term `(float` τ`)` denotes the floating point number equal to the object denoted by τ.

`float-digits` - The term `(float-digits` τ`)` denotes the number of digits used in the representation of a floating point number denoted by τ.

`float-precision` - The term `(float-precision` τ`)` denotes the number of significant digits in the floating point number denoted by τ.

`float-radix` - The term `(float-radix` τ`)` denotes the radix of the floating point number denoted by τ. The most common values are 2 and 16.

`float-sign` - The term `(float-sign` τ_1 τ_2`)` denotes a floating-point number with the same sign as the object denoted by τ_1 and the same absolute value as the object denoted by τ_2.

`floor` - The term `(floor` τ`)` denotes the largest integer less than the object denoted by τ.

`fround` - The term `(fround` τ`)` is equivalent to `(ffloor (+ 0.5` τ`))`.

`ftruncate` - The term `(ftruncate` τ`)` denotes the largest integer (as a floating point number) less than the object denoted by τ.

`gcd` - The term `(gcd` $\tau_1 \ldots \tau_n$`)` denotes the greatest common divisor of the objects denoted by τ_1 through τ_n.

`imagpart` - The term `(imagpart` τ`)` denotes the imaginary part of the object denoted by τ.

`integer-decode-float` - The term `(integer-decode-float` τ`)` denotes the significand of the object denoted by τ.

`integer-length` - The term `(integer-length` τ`)` denotes the number of bits required to store the absolute magnitude of the object denoted by τ.

`isqrt` - The term `(isqrt` τ`)` denotes the integer square root of the object denoted by τ.

`lcm` - The term `(lcm` $\tau_1 \ldots \tau_n$`)` denotes the least common multiple of the objects denoted by $\tau_1, \ldots, \tau_n$.

`log` - The term `(log` τ_1 τ_2`)` denotes the logarithm of the object denoted by τ_1 in the base denoted by τ_2.

`logand` - The term `(logand` $\tau_1 \ldots \tau_n$`)` denotes the bit-wise logical and of the objects denoted by τ_1 through τ_n.

`logandc1` - The term `(logandc1` τ_1 τ_2`)` is equivalent to `(logand (lognot` τ_1`)` τ_2`)`.

`logandc2` - The term `(logandc2` τ_1 τ_2`)` is equivalent to `(logand` τ_1 `(lognot` τ_2`))`.

`logcount` - The term `(logcount` τ`)` denotes the number of *on* bits in the object denoted by τ. If the denotation of τ is positive, then the one bits are counted; otherwise, the zero bits in the twos-complement representation are counted.

`logeqv` - The term `(logeqv` $\tau_1 \ldots \tau_n$`)` denotes the logical-exclusive-or of the objects denoted by $\tau_1, \ldots, \tau_n$.

`logior` - The term `(logior` $\tau_1 \ldots \tau_n$`)` denotes the bit-wise logical inclusive or of the objects denoted by τ_1 through τ_n. It denotes 0 if there are no arguments.

`lognand` - The term `(lognand` τ_1 τ_2`)` is equivalent to `(lognot (logand` τ_1 τ_2`))`.

`lognor` - The term `(lognor` τ_1 τ_2`)` is equivalent to `(not (logior` τ_1 τ_2`))`.

`lognot` - The term `(lognot` τ`)` denotes the bit-wise logical not of the object denoted by τ.

`logorc1` - The term `(logorc1` τ_1 τ_2`)` is equivalent to `(logior (lognot` τ_1`)` τ_2`)`.

`logorc2` - The term `(logorc2` τ_1 τ_2`)` is equivalent to `(logior` τ_1 `(lognot` τ_2`))`.

`logxor` - The term `(logxor` $\tau_1 \ldots \tau_n$`)` denotes the bit-wise logical exclusive or of the objects denoted by τ_1 through τ_n. It denotes 0 if there are no arguments.

`max` - The term `(max` $\tau_1 \ldots \tau_k$`)` denotes the largest object denoted by τ_1 through τ_n.

`min` - The term `(min` $\tau_1 \ldots \tau_k$`)` denotes the smallest object denoted by τ_1 through τ_n.

`mod` - The term `(mod` τ_1 τ_2`)` denotes the root of the object denoted by τ_1 modulo the object denoted by τ_2. The result will have the same sign as denoted by τ_1.

`numerator` - The term `(numerator` τ`)` denotes the numerator of the canonical reduced form of the object denoted by τ.

`phase` - The term `(phase` τ`)` denotes the angle part of the polar representation of the object denoted by τ (in radians).

`rationalize` - The term `(rationalize` τ`)` denotes the rational representation of the object denoted by τ.

`realpart` - The term `(realpart` τ`)` denotes the real part of the object denoted by τ.

`rem` - The term `(rem <number> <divisor>)` denotes the remainder of the object denoted by `<number>` divided by the object denoted by `<divisor>`. The result has the same sign as the object denoted by `<divisor>`.

`round` - The term `(round` τ`)` denotes the integer nearest to the object denoted by τ. If the object denoted by τ is halfway between two integers (for example 3.5), it denotes the nearest integer divisible by 2.

`scale-float` - The term `(scale-float` τ_1 τ_2`)` denotes a floating-point number that is the representational radix of the object denoted by τ_1 raised to the integer denoted by τ_2.

`signum` - The term `(signum` τ`)` denotes the sign of the object denoted by τ. This is one of -1, 1, or 0 for rational numbers, and one of -1.0, 1.0, or 0.0 for floating point numbers.

`sin` - The term `(sin` τ`)` denotes the sine of the object denoted by τ (in radians).

`sinh` - The term `(sinh` τ`)` denotes the hyperbolic sine of the object denoted by τ (in radians).

`sqrt` - The term `(sqrt` τ`)` denotes the principal square root of the object denoted by τ.

`tan` - The term `(tan` τ`)` denotes the tangent of the object denoted by τ (in radians).

`tanh` - The term `(tanh` τ`)` denotes the hyperbolic tangent of the object denoted by τ (in radians).

`truncate` - The term `(truncate` τ`)` denotes the largest integer less than the object denoted by τ.

§5.2 Relations on Numbers

`integer` - The sentence `(integer` τ`)` means that the object denoted by τ is an integer.

`real-number` - The sentence `(real-number` τ`)` means that the object denoted by τ is a real number.

`complex-number` - The sentence `(complex-number` τ`)` means that the object denoted by τ is a complex number.

```
(defrelation number (?x) :=
  (or (real-number ?x) (complex-number ?x)))
```
(5.6)

```
(defrelation natural (?x) :=  (and (integer ?x) (>= ?x 0)))
```
(5.7)

```
(defrelation rational-number (?x) :=
  (exists (?y) (and (integer ?y) (integer (* ?x ?y)))))
```
(5.8)

`<` - The sentence `(<` τ_1 τ_2`)` is true if and only if the number denoted by τ_1 is less than the number denoted by τ_2.

```
(defrelation > (?x ?y) := (< ?y ?x))
```
(5.9)

```
(defrelation =< (?x ?y) := (or (= ?x ?y) (< ?x ?y)))
```
(5.10)

```
(defrelation >= (?x ?y) := (or (> ?x ?y) (= ?x ?y)))
```
(5.11)

```
(defrelation positive (?x) := (> ?x 0))
```
(5.12)

```
(defrelation negative (?x) := (< ?x 0))
```
(5.13)

```
(defrelation zero (?x) := (= ?x 0))
```
(5.14)

```
(defrelation odd-integer (?x) := (integer (/ (+ ?x 1) 2))
```
(5.15)

```
(defrelation even-integer (?x) := (integer (/ ?x 2))
```
(5.16)

`logbit` - The sentence `(logbit` τ_1 τ_2`)` is true if bit τ_2 of τ_1 is 1.

`logtest` - The sentence `(logtest` τ_1 τ_2`)` is true if the logical *and* of the two's-complement representation of the integers τ_1 and τ_2 is not zero.

Chapter 6
Lists

A *list* is a finite sequence of objects. The objects in a list need not be KIF expressions, though they may be. In other words, it is just as acceptable to talk about a list of two people as it is to talk about a list of two symbols.

In KIF, we use the term `(listof` τ_1 `...` τ_k`)` to denote the list of objects denoted by τ_1, ..., τ_k. For example, the following expression denotes the list of an object named `mary`, a list of objects named `tom`, `dick`, and `harry`, and an object named `sally`.

```
(listof mary (listof tom dick harry) sally)
```

The relation `list` is the type predicate for lists. An object is a list if and only if there is a corresponding expression involving the `listof` operator.

```
(defrelation list (?x) :=
  (exists (@l) (= ?x (listof @l)))                              (6.1)
```

The object constant `nil` denotes the empty list. `null` tests whether or not an object is the empty list. The relation constants `single`, `double`, and `triple` allow us to assert the length of lists containing one, two, and three elements, respectively.

```
(defobject nil := (listof))                                     (6.2)

(defrelation null (?l) := (= ?l (listof)))                      (6.3)

(defrelation single (?l) := (exists ?x (= ?l (listof ?x))))     (6.4)

(defrelation double (?l) :=
  (exists (?x ?y) (= ?l (listof ?x ?y))))                       (6.5)

(defrelation triple (?l) :=
  (exists (?x ?y ?z) (= ?l (listof ?x ?y ?z))))                 (6.6)
```

The functions `first`, `rest`, `last`, and `butlast` each take a single list as argument and select individual items or sublists from those lists.

```
(deffunction first (?l) := (if (= (listof ?x @items) ?l) ?x)    (6.7)

(deffunction rest (?l) :=
  (cond ((null ?l) ?l)
        ((= ?l (listof ?x @items)) (listof @items))))           (6.8)

(deffunction last (?l) :=
  (cond ((null ?l) bottom)
        ((null (rest ?l)) (first ?l))
        (true (last (rest ?l)))))                               (6.9)
```

```
(deffunction butlast (?l) :=
  (cond ((null ?l) bottom)
        ((null (rest ?l)) nil)
        (true (cons (first ?l) (butlast (rest ?l))))))        (6.10)
```

The sentence (`item` τ_1 τ_2) is true if and only if the object denoted by τ_2 is a non-empty list and the object denoted by τ_1 is either the first item of that list or an item in the rest of the list.

```
(defrelation item (?x ?l) :=
  (and (list ?l)
       (not (null ?l))
       (or (= ?x (first ?l)) (item ?x (rest ?l)))))        (6.11)
```

The sentence (`sublist` τ_1 τ_2) is true if and only if the object denoted by τ_1 is a final segment of the list denoted by τ_2.

```
(defrelation sublist (?l1 ?l2) :=
  (and (list ?l1)
       (list ?l2)
       (or (= ?l1 ?l2)
           (sublist ?l1 (rest ?l2)))))        (6.12)
```

The function `cons` adds the object specified as its first argument to the front of the list specified as its second argument.

```
(deffunction cons (?x ?l) :=
  (if (= ?l (listof @l)) (listof ?x @l)))        (6.13)
```

The function `append` adds the items in the list specified as its first argument to the list specified as its second argument. The function `revappend` is simiar, except that it adds the items in reverse order.

```
(deffunction append (?l1 ?l2) :=
  (if (null ?l1) (if (list ?l2) ?l2)
      (cons (first ?l1) (append (rest ?l1) ?l2))))        (6.14)
```

```
(deffunction revappend (?l1 ?l2) :=
  (if (null ?l1) (if (list ?l2) ?l2)
      (revappend (rest ?l1) (cons (first ?l1) ?l2))))        (6.15)
```

The function `reverse` produces a list in which the order of items is the reverse of that in the list supplied as its single argument.

```
(deffunction reverse (?l) := (revappend ?l (listof)))        (6.16)
```

The functions `adjoin` and `remove` construct lists by adding or removing objects from the lists specified as their arguments.

```
(deffunction adjoin (?x ?l) := (if (item ?x ?l) ?l (cons ?x ?l))) (6.17)
```

```
(deffunction remove (?x ?l) :=
  (cond ((null ?l) nil)
        ((and (= ?x (first ?l)) (listp ?l))
         (remove ?x (rest ?l)))
        ((list ?l) (cons ?x (remove ?x (rest ?l))))))        (6.18)
```

The value of subst is the object or list obtained by substituting the object supplied as first argument for all occurrences of the object supplied as second argument in the object or list supplied as third argument.

```
(deffunction subst (?x ?y ?z) :=
  (cond ((= ?y ?z) ?x)
        ((null ?z) nil)
        ((list ?z) (cons (subst ?x ?y (first ?z))
                         (subst ?x ?y (rest ?z))))
        (true ?z)))                                          (6.19)
```

The function constant length gives the number of items in a list. nth returns the item in the list specified as its first argument in the position specified as its second argument. nthrest returns the list specified as its first argument minus the first n items, where n is the number specified as its second argument.

```
(deffunction length (?l) :=
   (cond ((null ?l) 0)
         ((list ?l) (1+ (length (rest ?l))))))               (6.20)

(deffunction nth (?l ?n) :=
   (cond ((= ?n 1) (first ?l))
         ((positive ?n) (nth (rest ?l) (1- ?n)))))           (6.21)

(deffunction nthrest (?l ?n) :=
  (cond ((= ?n 0) (if (listp ?l) ?l))
        ((positive ?n)) (nthrest (rest ?l) (1- ?n)))))       (6.22)
```

Chapter 7
Sets

In many applications, it is helpful to talk about sets of objects as objects in their own right, e.g. to specify their cardinality, to talk about subset relationships, and so forth.

The formalization of sets of simple objects is a simple matter; but, when we begin to talk about sets of sets, the job becomes difficult due to the threat of paradoxes (like Russell's hypothesized set of all sets that do not contain themselves).

Fortunately, there is no shortage of mathematical theories for our use in KIF – various higher order logics, Zermelo-Fraenkel set theory, von Neuman-Bernays-Gödel set theory, Quine's variants on the previous two approaches, the more recently elaborated proposals by Feferman and Aczel, and so forth. In KIF, we have adopted a version of the von Neumann-Bernays-Gödel set theory.

In our presentation here, we first discuss the basic concepts of this theory – the notions of set and membership. Next, we look at some terminology for describing the properties of sets. We then present the standard axioms of set theory. Finally, we discuss the threat of paradox and indicate how our use of the von Neumann-Bernays-Gödel set theory avoids this problem.

An important word of warning for mathematicians. In KIF, certain words are used nontraditionally. Specifically, the standard notion of *class* is here called a *set*; the standard notion of *set* is replaced by the notion of *bounded* set; and the standard notion of *proper class* is replaced by *unbounded* set.

§7.1 Basic Concepts

In KIF, a fundamental distinction is drawn between *individuals* and *sets*. A set is a collection of objects. An individual is any object that is not a set.

A distinction is also drawn between objects that are *bounded* and those that are *unbounded*. This distinction is orthogonal to the distinction between individuals and sets. There are bounded individuals and unbounded individuals. There are bounded sets and unbounded sets.

The fundamental relationship among these various types of entities is that of membership. Sets can have members, but individuals cannot. Bounded objects can *be* members of sets, but unbounded objects cannot. (It is this condition that allows us to avoid the traditional paradoxes of set theory.)

In KIF, we use the unary relation constants `individual` and `set`, `bounded` and `unbounded` to make these distinctions; and we use the binary relation constant `member` to talk about membership.

The sentence (`individual` τ) is true if and only if the object denoted by τ is an individual. The sentence (`set` τ) is true if and only if the object denoted by τ is a set. As just described, individuals and sets are exhaustive and mutually disjoint.

`(or (set ?x) (individual ?x))` (7.1)

`(or (not (set ?x)) (not (individual ?x)))` (7.2)

The sentence `(bounded` τ`)` is true if and only if the object denoted by τ is bounded. The sentence `(unbounded` τ`)` is true if and only if the object denoted by τ is unbounded. Boundedness and unboundedness are exhaustive and mutually disjoint.

```
(or (bounded ?x) (unbounded ?x))                              (7.3)
```

```
(or (not (bounded ?x)) (not (unbounded ?x)))                  (7.4)
```

The sentence `(member` τ_1 τ_2`)` is true if and only if the object denoted by τ_1 is contained in the set denoted by τ_2. As mentioned above, an object can be a member of another object if and only if the former is bounded and the latter is a set.

```
(=> (member ?x ?s)
    (bounded ?x))                                             (7.5)
```

```
(=> (member ?x ?s)
    (set ?x))                                                 (7.6)
```

An important property shared by all sets is the extensionality property. Two sets are identical if and only if they have the same members.

```
(=> (and (set ?s1) (set ?s2))
    (<=> (forall (?x) (<=> (member ?x ?s1) (member ?x ?s2)))
         (= ?s1 ?s2)))                                        (7.7)
```

§7.2 Sets

To allow us to name specific sets, KIF provides the operators `setof` and `setofall`.

The term `(setof` τ_1 `...` τ_k`)` denotes the set consisting of the objects denoted by τ_1, ..., τ_k that are bounded.

```
(=> (item ?x (listof @items))
    (bounded ?x)
    (member ?x (setof @items)))                               (7.8)
```

```
(=> (member ?x (setof @items))
    (item ?x (listof @items)))                                (7.9)
```

Note that the cardinality of the set denoted by `(setof` τ_1 `...` τ_k`)` can be less than k. By definition, an object can appear in a set only once. Consequently, if τ_i and τ_j (for different i and j) denote the same object, the resulting set must contain fewer than k members.

The operator `setofall` allows us to define sets in terms of their properties. The term `(setofall` τ ϕ`)` denotes the set of all bounded objects denoted by τ for any assignment of the free variables in τ that satisfies ϕ.

`(<=> (member` τ `(setofall` ν ϕ`))`
`     (and (bounded` τ`)` $\phi_{\nu/\tau}$`))` (7.10)

Note that the first argument to `setofall` must be a term, not a list of variables as with `forall` and `exists`. The term can be a single variable, a functional expression, or even a quantified term. If the term contains no free variables, then the set consists of either zero members or one member, depending on the truth value of the embedded sentence.

The `empty` relation is true of the empty set but false of all other objects.

```
(defrelation empty (?x) := (= ?x (setof)))                        (7.11)
```

In KIF, the functions `union`, `intersection`, `difference`, and `complement` are defined as follows.

```
(deffunction union (@sets) :=
  (if (forall (?s) (=> (item ?s (listof @sets)) (set ?s)))
      (setofall ?x (exists (?s) (and (item ?s (listof @sets))
                                      (member ?x ?s))))))         (7.12)

(deffunction intersection (@sets) :=
  (if (forall (?s) (=> (item ?s (listof @sets)) (set ?s)))
      (setofall ?x (forall (?s) (=> (item ?s (listof @sets))
                                     (member ?x ?s))))))          (7.13)

(deffunction difference (?set @sets) :=
  (if (and (set ?set)
           (forall (?s) (=> (item ?s (listof @sets)) (set ?s))))
      (setofall ?x
                (and (member ?x ?set)
                     (forall (?s) (=> (item ?s (listof @sets))
                                      (not (member ?x ?s)))))))))  (7.14)

(deffunction complement (?s) :=
  (if (set ?s)
      (setofall ?x (not (member ?x ?s)))))                        (7.15)
```

The functions `generalized-union` and `generalized-intersection` allow us to talk about the union and intersection of the sets in a set of sets.

```
(deffunction generalized-union (?set) :=
  (if (and (set ?set)
           (forall (?s) (=> (member ?s ?set) (set ?s)))
      (setofall ?x (exists (?s) (and (member ?s ?set)
                                      (member ?x ?s))))))         (7.16)

(deffunction generalized-intersection (?set) :=
  (if (and (set ?set)
           (forall (?s) (=> (member ?s ?set) (set ?s)))
      (setofall ?x (exists (?s) (=> (member ?s ?set)
                                     (member ?x ?s))))))          (7.17)
```

The sentence (`subset` τ_1 τ_2) is true if and only if τ_1 and τ_2 are sets and the objects in the set denoted by τ_1 are contained in the set denoted by τ_2.

```
(defrelation subset (?s1 ?s2) :=
  (and (set ?s1) (set ?s2)
       (forall ?x (=> (member ?x ?s1) (member ?x ?s2)))))          (7.18)
```

The sentence (`proper-subset` τ_1 τ_2) is true if the set denoted by τ_1 is a subset of the set denoted by τ_2 but not vice-versa.

```
(defrelation proper-subset (?s1 ?s2) :=
  (and (subset ?s1 ?s2)
       (not (subset ?s2 ?s1))))                                     (7.19)
```

Two sets are disjoint if and only if there is no object that is a member of both sets. Sets are pairwise-disjoint if and only if every set is disjoint from every other set. Sets are mutually-disjoint if and only if there is no object that is a member of all of the sets. Note that mutually-disjoint sets need not be pairwise disjoint; in fact, every pair of sets might be overlapping. For example, the sets $\{a, b\}$ and $\{b, c\}$ and $\{a, c\}$ are mutually disjoint but not pairwise disjoint.

```
(defrelation disjoint (?s1 ?s2) :=
  (empty (intersection ?s1 ?s2)))                                   (7.20)

(defrelation pairwise-disjoint (@sets) :=
  (forall (?s1 ?s2) (=> (item ?s1 (listof @sets))
                        (item ?s2 (listof @sets))
                        (or (= ?s1 ?s2) (disjoint ?s1 ?s2)))))      (7.21)

(defrelation mutually-disjoint (@sets) :=
  (= (intersection @sets) (set)))                                   (7.22)

(defrelation set-partition (?s @sets) :=
  (and (= ?s (union @sets))
       (pairwise-disjoint @sets)))                                  (7.23)

(defrelation set-cover (?s @set) :=
  (subset ?s (union @sets)))
                                                                    (7.24)
```

We close this section with two axioms that allow us to conclude that sets of various sorts do, in fact, exist. The first is the *axiom of regularity* – every non-empty set has an element with which it has no members in common.

```
(forall (?s)
  (=> (not (empty ?s))
      (exists (?u) (and (member ?u ?s) (disjoint ?u ?s)))))        (7.25)
```

This axiom is not absolutely essential for set theory. However, it makes many proofs a lot easier, and so it is commonly included among the axioms of set theory.

The second axiom is the *axiom of choice*. It asserts that there is a set that associates every bounded set with a distinguished element of that set. In effect, it *chooses* an element from every bounded set.

```
(exists (?s)
  (and (set ?s)
       (forall (?x) (=> (member ?x ?s) (double ?x)))
       (forall (?x ?y ?z) (=> (and (member (listof ?x ?y) ?s)
                                   (member (listof ?x ?z) ?s))
                              (= ?y ?z)))
       (forall (?u)
         (=> (and (bounded ?u) (not (empty ?u)))
             (exists (?v) (and (member ?v ?u)
                               (member (listof ?u ?v) ?s))))))))  (7.26)
```

Again, this axiom is not essential. In some versions of set theory, the axiom of choice is omitted. However, it is a highly desirable property and is included here for that reason.

§7.3 Boundedness

As mentioned earlier, the key difference between bounded and unbounded objects is that the former can be members of other sets while the latter cannot. This fact establishes a necessary and sufficient test for boundedness – an object is bounded just in case it is a member of a set. However, this is not very helpful, since we often need to determine whether or not an object is bounded based on other properties, not the sets of which it is a member. In this section, we look at some axioms that help us make this determination for sets.

First of all, we have the *finite set axiom*. Any finite set of bounded sets is itself a bounded set.

```
(bounded (setof @l))                                               (7.27)
```

The *subset axiom* assures that the set of all of subsets of a bounded set is also a bounded set.

```
(=> (bounded ?v) (bounded (setofall ?u (subset ?u ?v))))           (7.28)
```

The *union axiom* tells us that the generalized union of any bounded set of bounded sets is also a bounded set. Since every finite set is bounded, this allows us to conclude, as a special case, that the union of any finite number of bounded sets is a bounded set.

```
(=> (and (bounded ?u) (forall (?x) (=> (member ?x ?u) (bounded ?x))))
    (bounded (generalized-union ?u)))                              (7.29)
```

The *intersection axiom* tells us that the intersection of a bounded set and any other set is a bounded set. So long as one of the sets defining the intersection is bounded, the resulting set is bounded.

```
(=> (and (bounded ?u) (set ?s))
    (bounded (intersection ?u ?s)))                                (7.30)
```

Finally, we have the *axiom of infinity*. There is a bounded set containing a set, a set that properly contains that set, a third set that properly contains the second set, and so forth. In short, there is at least one bounded set of infinite cardinality.

```
(exists (?u)
  (and (bounded ?u)
       (not (empty ?u))
       (forall (?x)
         (=> (member ?x ?u)
             (exists (?y) (and (member ?y ?u)
                               (proper-subset ?x ?y)))))))          (7.31)
```

§7.4 Paradoxes

The presence of sets in our universe of discourse does not in itself lead to paradoxes. The paradoxes appear only when we try to define set primitives that are too powerful. We have defined the sentence (member τ σ) to be true in exactly those cases when the object denoted by τ is a member of the set denoted by σ, and we might consider defining the term (setofall τ ϕ) to mean simply the set of all objects denoted by τ for any assignment of the free variables of τ that satisfies ϕ. Unfortunately, these two definitions quickly lead to paradoxes.

Let $\phi_{\nu/\tau}$ be the result of substituting term τ for all free occurrences of ν in sentence ϕ. Provided that τ is a term not containing any free variables captured in $\phi_{\nu\tau}$, then the following schema follows from our informal definition. This schema is called the *principle of unrestricted set abstraction.*

(<=> (member τ (setofall ν ϕ)) $\phi_{\nu/\tau}$)

Now, let us substitute the variable ?x for ν, the sentence (not (member ?x ?x)) for ϕ, and the term (setofall ?x (not (member ?x ?x))) for τ. The resulting instance of the principle of unrestricted set abstraction follows.

```
(<=> (member (setofall ?x (not (member ?x ?x)))
             (setofall ?x (not (member ?x ?x))))
     (not (member (setofall ?x (not (member ?x ?x)))
                  (setofall ?x (not (member ?x ?x))))))
```

This sentence has the form (<=> ϕ (not ϕ)), which cannot be true in any interpretation. This is Russell's paradox, only one of a family of familiar absurdities following from the principle of unrestricted set abstraction.

It is crucial that the paradoxes of set theory be avoided. One of the goals in the design of KIF is that it have a clearly specified model-theoretic semantics in terms of which the concepts of entailment, equivalence, consistency, soundness and completeness can be defined. If the paradoxes are allowed to persist in principle, even if they are easy to avoid in practice, the consequence would be that no KIF theory would be true in any model. Definitions couched in terms of models would be trivialized, becoming useless. All sentences would be entailed by any theory, any two theories would be equivalent, no theory would be consistent, every possible inference rule would be sound, and so on.

In the von-Neuman-Gödel-Bernays version of set theory, these paradoxes are avoided by replacing the principle of unrestricted set abstraction with the *principle of restricted set abstraction* given above.

$$
\begin{array}{l}
\texttt{(<=> (member } \tau \texttt{ (setofall } \nu \ \phi\texttt{))} \\
\quad\quad \texttt{(and (bounded } \tau\texttt{) } \phi_{\nu/\tau}\texttt{))}
\end{array}
$$

With this principle, there are two reasons why something may be excluded from a set `(setofall` ν ϕ`)`. It may fail to be a member because it does not satisfy the defining condition ϕ, or it may be excluded because it is an unbounded object. Conditioning the membership of objects in this set on their boundedness effectively eliminates the paradoxes.

Chapter 8
Functions and Relations

In KIF, we can describe specific functions and relations by naming them with function constants and relation constants and then writing sentences in which those names occur in functional or relational position. For most purposes, this is adequate; but in some cases it is also useful to describe functions and relations more generally – to name their properties (such as associativity and transitivity) and to write axioms relating these properties (possibly quantifying over the functions and relations possessing these properties). In order to do this, we need to treat functions and relations as objects in our universe of discourse.

By definition, functions and relations are sets of lists of objects from our universe of discourse. The immediately preceding chapters offer a vocabulary for describing lists and sets in general. However, functions and relations have enough special properties to warrant additional vocabulary.

In what follows, we begin by presenting the KIF vocabulary for abstraction and application of functions and relations. We then talk about the use of functions and relation constants in argument position of terms. Finally, we present some supporting vocabulary.

Note that the introduction of functions and relations into our universe of discourse comes with the threat of paradox, as with sets in general. In KIF, we sidestep such paradoxes by defining the sets comprising our functions and relations in terms of the set concepts introduced in the preceding chapter.

§8.1 Basic Vocabulary

As described in chapter 3, a relation is an arbitrary set of lists. A collection of objects satisfies a relation if and only if the list of those objects is a member of this set.

```
(defrelation relation (?r) :=
  (and (set ?r)
       (forall (?x) (=> (member ?x ?r) (list ?x)))))                (8.1)
```

Since KIF allows for n-ary relations, the lists in the set need not be of the same length. For example, the $<$ relation is defined on 2-lists, 3-lists, 4-lists, and so forth.

A function is a set of lists in which the items in every list except for the last determine the last item, i.e. there cannot be two lists that agree on all but the last item and disagree on the last item.

```
(defrelation function (?f) :=
  (and (relation ?f)
       (forall (?l ?m)
         (=> (member ?l ?f)
             (member ?m ?f)
             (= (butlast ?l) (butlast ?m))
             (= (last ?l) (last ?m))))))                              (8.2)
```

As with relations in general, the lists of a function need not be of the same length, to allow for functions of variable arity. For example, associative functions like $+$ and $*$ functions can be applied to arbitrary numbers of arguments.

An important difference between our treatment of functions and the traditional treatment is that functions need not contain lists for every possible combination of arguments. If a function is undefined for a particular combination of objects (i.e. if its value is $\perp$), then we omit that list from the set. Thus, even though our universe of discourse is infinite, it is possible for a function to have a finite number of lists.

§8.2 Function and Relation Constants

Since function constants and relation constants denote functions and relations and since functions and relations are objects in our universe of discourse, it is natural to allow function and relation constants to appear as as arguments in terms and sentences.

Unfortunately, in order to avoid paradoxes, it is sometimes essential for there to be a difference between the interpretation of a function or relation constant and its semantic value. We can sidestep these potential difficulties by writing axioms that define function and relation constants, used in argument position, in terms of the `setof` operator.

As described in chapter 4, the semantic value of a function constant π is the set of lists of objects corresponding to the function denoted by π. The following axiom schema expresses this property.

$$\texttt{(= } \pi \texttt{ (setofall (listof } \nu_1 \ \ldots \ \nu_k \ \nu\texttt{) (= (}\pi \ \nu_1 \ \ldots \ \nu_k\texttt{) } \nu\texttt{))} \tag{8.3}$$

Similarly, the semantic value of a relation constant ρ is the set of lists of objects that satisfy the relation denoted by ρ. Again, we have an axiom schema corresponding to this property.

$$\texttt{(= } \rho \texttt{ (setofall (listof } \nu_1 \ \ldots \ \nu_k\texttt{) (}\rho \ \nu_1 \ \ldots \ \nu_k\texttt{)))} \tag{8.4}$$

The use of function and relation constants in argument position weakens the distinction between object constants on the one hand and function and relation constants on the other.

The distinction between function and relation constants can also be weakened, since functions are a special class of relations. Any position that requires a relation constant can also be filled by a function constant. When this happens, the function denoted by the function constant is treated as a relation (which it is). For instance, in the following sentence, the first occurrence of `+` plays the role of a relation constant, while in the second occurrence, it plays the role of a function constant. (In both cases, `+` denotes the same entity.)

```
(and (+ 2 3 5)
     (= (+ 2 3 5) 10))
```

In KIF, all function constants are treated as relation constants, and all relation constants (and hence all function constants) are treated as object constants. An object constant is still prohibited from occurring as the first item of a term or a sentence, and a relation constant that is not a function constant cannot occupy the first position in a term.

The convenience afforded by the ability to use function and relation constants as arguments and to use function constants in relational position often causes concern over grammatical ambiguity. The expression `(+ 5 2 3)` is both a term and a sentence. Fortunately, this ambiguity is always resolved when such expressions occur within well-formed databases. Any expression that occurs at top level cannot be a term. An expression embedded in a non-operator expression must be a term. An expression embedded in an operator expression can be either a term or a sentence, but in either case the type of the expression is known from the operator's syntax.

§8.3 Concretion

If τ denotes a relation, then the sentence (holds τ τ_1 ... τ_k) is true if and only if the list of objects denoted by τ_1,...,τ_k is a member of that relation.

```
(defrelation holds (?r @args) :=
  (and (relation ?r) (member (listof @args) ?r)))
```
(8.5)

If τ denotes a function with a value for the objects denoted by τ_1,..., τ_k, then the term (value τ τ_1 ... τ_k) denotes the value of applying that function to the objects denoted by τ_1,...,τ_k. Otherwise, the value is undefined.

```
(deffunction value (?f @args) :=
  (if (and (function ?f)
           (member ?l ?f)
           (= (butlast ?l) (listof @args)))
      (last ?l)))
```
(8.6)

```
(deffunction apply (?f ?l) :=
  (if (and (function ?f) (= ?l (listof @args)))
      (value ?f @args)))
```
(8.7)

```
(deffunction map (?f ?l) :=
  (if (null ?l) (list)
      (cons (value ?f (first ?l)) (map ?f (rest ?l)))))
```
(8.8)

§8.4 Abstraction

As described in chapter 4, the semantic value of the term (lambda (ν_1 ... ν_k [ω]) τ) is the set of lists associated with the function that maps every list of objects "matching" the variable list to the value of τ when the variables in the variable list are assigned to the objects in the list. We can capture this meaning with the following axiom schema.

(= (lambda (ν_1 ... ν_k [ω]) τ)
   (setofall (listof ν_1 ... ν_k [ω] ν) (= ν τ)))   (8.9)

The semantic value of the term (kappa (ν_1 ... ν_k [ω]) ϕ) is the set of lists associated with the relation that holds of every list of objects "matching" the variable list for which the sentence ϕ is satisfied. We can capture this meaning with the following axiom schema.

$$\texttt{(= (kappa (}\nu_1\ \texttt{...}\ \nu_k\ \texttt{[}\omega\texttt{]) }\phi\texttt{)}$$
$$\texttt{(setofall (listof }\nu_1\ \texttt{...}\ \nu_k\ \texttt{[}\omega\texttt{]) }\phi\texttt{)))} \quad (8.10)$$

§8.5 Additional Concepts

The universe of a relation is the set of all objects occurring in some list contained in that relation.

```
(deffunction universe (?r) :=
  (if (relation ?r)
      (setofall ?x (exists (?l) (and (member ?l ?r)
                                     (item ?x ?l))))))          (8.11)
```

A unary relation is a non-empty relation in which all lists have exactly one item.

```
(defrelation unary-relation (?r) :=
  (and (not (empty ?r))
       (forall (?l) (=> (member ?l ?r) (single ?l)))))          (8.12)
```

A binary relation is a non-empty relation in which all lists have exactly two items. The inverse of a binary relation is a binary relation with all tuples reversed. The composition of a binary relation r_1 and a binary relation r_2 is a binary relation in which an object x is related to an object z if and only if there is an object y such that x is related to y by r_1 and y is related to z by r_2.

```
(defrelation binary-relation (?r) :=
  (and (not (empty ?r))
       (forall (?l) (=> (member ?l ?r) (double ?l)))))          (8.13)

(deffunction inverse (?r) :=
  (if (binary-relation ?r)
      (setofall (listof ?y ?x) (holds ?r ?x ?y))))              (8.14)

(deffunction composition (?r1 ?r2) :=
  (if (and (binary-relation ?r1)
           (binary-relation ?r2))
      (setofall (listof ?x ?z)
        (exists (?y)
          (and (holds ?r1 ?x ?y)
               (holds ?r2 ?y ?z))))))                           (8.15)

(defrelation one-one (?r) :=
  (and (binary-relation ?r)
       (function ?r)
       (function (inverse ?r))))                                (8.16)
```

```
(defrelation many-one (?r) :=
  (and (binary-relation ?r)
       (function ?r)))                                        (8.17)
```

```
(defrelation one-many (?r) :=
  (and (binary-relation ?r)
       (function (inverse ?r))))                              (8.18)
```

```
(defrelation many-many (?r) :=
  (and (binary-relation ?r)
       (not (function ?r))
       (not (function (inverse ?r)))))                        (8.19)
```

A unary function is a function with a single argument and a single value. Hence, it is also a binary relation.

```
(defrelation unary-function (?f) :=
  (and (function ?f)
       (binary-relation ?f)))                                 (8.20)
```

A binary function is a function with two arguments and one value. Hence, it is a relation with three arguments.

```
(defrelation binary-function (?f) :=
  (and (function ?f)
       (not (empty ?f))
       (forall (?l) (=> (member ?l ?f) (triple ?l)))))        (8.21)
```

Chapter 9
Metaknowledge

§9.1 Naming Expressions

In formalizing knowledge about knowledge, we use a conceptualization in which expressions are treated as objects in the universe of discourse and in which there are functions and relations appropriate to these objects. In our conceptualization, we treat atoms as primitive objects (i.e. having no subparts). We conceptualize complex expressions (i.e. non-atoms) as lists of subexpressions (either atoms or other complex expressions). In particular, every complex expression is viewed as a list of its immediate subexpressions.

For example, we conceptualize the sentence `(not (p (+ a b c) d))` as a list consisting of the operator `not` and the sentence `(p (+ a b c) d)`. This sentence is treated as a list consisting of the relation constant `p` and the terms `(+ a b c)` and `d`. The first of these terms is a list consisting of the function constant `+` and the object constants `a`, `b`, and `c`.

For Lisp programmers, this conceptualization is relatively obvious, but it departs from the usual conceptualization of formal languages taken in the mathematical theory of logic. It has the disadvantage that we cannot describe certain details of syntax such as parenthesization and spacing (unless we augment the conceptualization to include string representations of expressions as well). However, it is far more convenient for expressing properties of knowledge and inference than string-based conceptualizations.

In order to assert properties of expressions in the language, we need a way of referring to those expressions. There are two ways of doing this in KIF.

One way is to use the `quote` operator in front of an expression. From the section on semantics, we know that a quotation denotes the expression embedded within the term. Therefore, to refer to the symbol `john`, we use the term `'john` or, equivalently, `(quote john)`. To refer to the expression `(p a b)`, we use the term `'(p a b)` or, equivalently, `(quote (p a b))`.

With a way of referring to expressions, we can assert their properties. For example, the following sentence ascribes to the individual named `john` the belief that the moon is made of a particular kind of blue cheese.

```
(believes john '(material moon stilton))
```

Note that, by nesting quotes within quotes, we can talk about quoted expressions. In fact, we can write towers of sentences of arbitrary heights, in which the sentences at each level talk about the sentences at the lower levels.

Since expressions are first-order objects, we can quantify over them, thereby asserting properties of whole classes of sentences. For example, we could say that Mary believes everything that John believes. This fact together with the preceding fact allows us to conclude that Mary also believes the moon to be made of blue cheese.

```
(=> (believes john ?p) (believes mary ?p))
```

The second way of referring to expressions is KIF is to use the `listof` function. For example, we can denote a complex expression like `(p a b)` by a term of the form `(listof 'p 'a 'b)`, as well as `'(p a b)`.

The advantage of the `listof` representation over the `quote` representation is that it allows us to quantify over parts of expressions. For example, let us say that Lisa is more skeptical than Mary. She agrees with John, but only on the composition of things. The first sentence below asserts this fact without specifically mentioning `moon` or `stilton`. Thus, if we were to later discover that John thought the sun to be made of chili peppers, then Lisa would be constrained to believe this as well.

```
(=> (believes john (listof 'material ?x ?y))
    (believes lisa (listof 'material ?x ?y)))
```

While the use of `listof` allows us to describe the structure of expressions in arbitrary detail, it is somewhat awkward. For example, the term `(listof 'material ?x ?y)` is somewhat awkward. Fortunately, we can eliminate this difficulty using backquote and comma. Rather than using the `listof` function constant as described above, we write the expression preceded by the backquote character ` and add a comma character , in front of any subexpression that is not to be taken literally. For example, we would rewrite the preceding sentence as follows.

```
(=> (believes john `(material ,?x ,?y))
    (believes lisa `(material ,?x ,?y)))
```

This approach is particularly nice in that it parallels the treatment of quoting and unquoting in Common Lisp. However, a warning is in order. All Common Lisps translate quoted expressions into lists with `quote` as the first element, e.g. `'(f a)` translates into `(quote (f a))`. However, not all Common Lisps are consistent in the handling of backquote. Some Lisps translate backquoted expressions into internal forms involving `listof`, e.g. `` `(f ,?x) `` translates into `(listof 'f ?x)`. Some use `cons`, e.g. `(cons 'f (cons ?x nil))`. Some use neither or a mixture. This does not prohibit our using the approach in KIF, but it means that we cannot rely on all Lisp readers to produce the internal form we want.

§9.2 Formalizing Syntax

In order to facilitate the encoding of knowledge about KIF, the language includes type relations for the various syntactic categories defined in chapter 2.

For every individual variable ν, there is an axiom asserting that it is indeed an individual variable. Each such axiom is a defining axiom for the `indvar` relation.

`(indvar (quote` ν`))` (9.1)

For every sequence variable ω, there is an axiom asserting that it is a sequence variable. Each such axiom is a defining axiom for the `seqvar` relation.

`(seqvar (quote` ω`))` (9.2)

```
(defrelation termop (?x) :=
```

```
  (member ?x (setof 'quote 'if 'cond 'the 'setof
                    'kappa 'lambda)))
```
(9.3)

```
(defrelation sentop (?x) :=
  (member ?x (setof 'not 'and 'or '=> '<= '<=> 'forall 'exists)))
```
(9.4)

```
(defrelation ruleop (?x) := (member ?x (setof  '=>> '<<=)))
```
(9.5)

```
(defrelation defop (?x) :=
  (member ?x (setof 'defobject 'deffunction 'defrelation ':=
                    ':=> ':axiom ':conservative-axiom)))
```
(9.6)

For every constant σ, there is an axiom asserting that it is a constant. Each such axiom is a defining axiom for the `constant` relation. The category of each constant is determined from its definition and/or the uses of the constant in a knowledge base.

```
(defrelation constant (constant (quote $\sigma$)))
```
(9.7)

From these basic vocabulary items, we define variables, operators, words, and expressions.

```
(defrelation variable (?x) := (or (indvar ?x) (seqvar ?x)))
```
(9.8)

```
(defrelation operator (?x) :=
  (or (termop ?x) (sentop ?x) (ruleop ?x) (defop ?x)))
```
(9.9)

```
(defrelation word (?x) :=
  (or (variable ?x) (operator ?x) (constant ?x)))
```
(9.10)

```
(defrelation expression (?x) :=
  (or (word ?x)
      (and (list ?x)
           (forall (?y) (=> (item ?y ?x) (expression ?y))))))
```
(9.11)

The sentence `(term` τ`)` is true if and only if the object denoted by τ is a term, i.e. it is either a constant, a variable, functional term, a list term, a set term, a quoted term, a logical term, or a quantified term.

```
(defrelation term (?x) :=
  (or (indvar ?x) (objconst ?x) (funconst ?x) (relconst ?x)
      (funterm ?x) (listterm ?x) (setterm ?x) (quoterm ?x)
      (logterm ?x) (quanterm ?x)))
```
(9.12)

```
(defrelation funterm (?x) :=
  (exists (?f ?tlist)
    (and (funconst ?f)
         (list ?tlist)
         (=> (item ?t  ?tlist) (term ?t))
```

```
        (= ?x (cons ?f ?tlist)))))                                     (9.13)

(defrelation listterm (?x) :=
  (exists ?tlist
    (and (list ?tlist)
         (=> (item ?t  ?tlist) (term ?t))
         (= ?x (cons 'listof ?tlist)))))                               (9.14)

(defrelation setterm (?x) :=
  (exists ?tlist
    (and (list ?tlist)
         (=> (item ?t  ?tlist) (term ?t))
         (= ?x (cons 'setof ?tlist)))))                                (9.15)

(defrelation (?x) :=
  (exists (?e)
    (and (expression ?e)
         (= ?x `(quote ,?e)))))                                        (9.16)

(defrelation logterm (?x) :=
  (or (exists (?p1 ?t1)
        (and (sentence ?p1) (term ?t1) (= ?x `(if ,?p1 ,?t1))))
      (exists (?p1 ?t1 ?t2)
        (and (sentence ?p1)
             (term ?t1)
             (term ?t2)
             (= ?x `(if ,?p1 ,?t1 ,?t2))))
      (exists ?clist
        (and (list ?clist)
             (=> (item ?c ?clist)
                 (exists (?p ?t)
                   (and (sentence ?p) (term ?t)
                        (= ?c (listof ?p ?t)))))
             (= ?x (cons 'cond ?clist))))))                            (9.17)

(defrelation quanterm (?x) :=
  (or (exists (?t ?p)
        (and (term ?t) (sentence ?p)
             (= ?x (listof 'the ?t ?p))))
      (exists (?t ?p)
        (and (term ?t) (sentence ?p)
             (= ?x (listof 'setof ?t ?p))))
      (exists (?vlist ?p)
        (and (list ?vlist) (sentence ?p)
             (>= (length ?vlist) 1)
```

```
          (=> (item ?v ?vlistp) (indvar ?v))
          (= ?x (listof 'kappa ?vlistp ?p))))
   (exists (?vlist ?sv ?p)
     (and (list ?vlist) (seqvar ?sv) (sentence ?p)
          (=> (item ?v ?vlist) (indvar ?v))
          (= ?x (listof 'kappa (append ?vlist (listof ?sv)) ?p))))
   (exists (?vlist ?t)
     (and (list ?vlist) (term ?t)
          (>= (length ?vlist) 1)
          (=> (item ?v ?vlistp) (indvar ?v))
          (= ?x (listof 'lambda ?vlistp ?t))))
   (exists (?vlist ?sv ?t)
     (and (list ?vlist) (seqvar ?sv) (sentence ?t)
          (=> (item ?v ?vlist) (indvar ?v))
          (= ?x (listof 'lambda
                        (append ?vlist (listof ?sv))
                        ?t))))))                          (9.18)
```

The sentence (`sentence` τ) is true if and only if the object denoted by τ is a sentence, i.e. it is either a logical constant, a relational sentence, a logical sentence, or a quantified sentence.

```
(defrelation sentence (?x) :=
  (or (logconst ?x) (relsent ?x) (equation ?x)
      (inequality ?x) (logsent ?x) (quantsent ?x)))       (9.19)

(defrelation relsent (?x) :=
  (exists (?r ?tlist)
    (and (or (relconst ?r) (funconst ?r)) (list ?tlist)
         (>= (length ?tlist) 1)
         (=> (item ?t ?tlist) (term ?t))
         (= ?x (cons ?r ?tlist)))))                       (9.20)

(defrelation equation (?x) :=
  (exists (?t1 ?t2)
    (and (term ?t1) (term ?t2)
         (= ?x `(= ,?t1 ,?t2)))))                         (9.21)

(defrelation inequality (?x) :=
  (exists (?t1 ?t2)
    (and (term ?t1) (term ?t2)
         (= ?x `(/= ,?t1 ,?t2)))))                        (9.22)

(defrelation logsent (?x) :=
  (or (negation ?x) (conjunction ?x) (disjunction ?x)
      (implication ?x) (reverse-implication ?x)
```

```
      (equivalence ?x)))                                          (9.23)

(defrelation negation (?x) :=
  (exists (?p)
    (and (sentence ?p)
         (= ?x `(not ,?p)))))                                     (9.24)

(defrelation conjunction (?x) :=
  (exists ?plist
    (and (list ?plist)
         (>= (length ?plist) 1)
         (=> (item ?p ?plist) (sentence ?p))
         (= ?x (cons 'and ?plist)))))                             (9.25)

(defrelation disjunction (?x) :=
  (exists ?plist
    (and (list ?plist)
         (>= (length ?plist) 1)
         (=> (item ?p ?plist) (sentence ?p))
         (= ?x (cons 'or ?plist)))))                              (9.26)

(defrelation implication (?x) :=
  (exists (?plist)
    (and (list ?plist)
         (>= (length ?plist) 2)
         (=> (item ?p ?plist) (sentence ?p))
         (= ?x (cons '=> ?plist)))))                              (9.27)

(defrelation reverse-implication (?x) :=
  (exists (?plist)
    (and (list ?plist)
         (>= (length ?plist) 2)
         (=> (item ?p ?plist) (sentence ?p))
         (= ?x (cons '<= ?plist)))))                              (9.28)

(defrelation equivalence (?x) :=
  (exists (?p1 ?p2)
    (and (sentence ?p1)
         (sentence ?p2)
         (= ?x `(<=> ,?p1 ,?p2)))))                               (9.29)

(defrelation quantsent (?x) :=
  (or (exists (?v ?p)
        (and (indvar ?v) (sentence ?p)
             (or (= ?x (listof 'forall ?v ?p))
```

```
              (= ?x (listof 'exists ?v ?p)))))
    (exists (?vlist ?p)
      (and (list ?vlist) (sentence ?p)
           (>= (length ?vlist) 1)
           (=> (item ?v ?vlist) (indvar ?v))
           (or (= ?x (listof 'forall ?vlist ?p))
               (= ?x (listof 'exists ?vlist ?p)))))))
```

(9.30)

§9.3 Changing Levels of Denotation

The vocabulary introduced in the preceding subsection allows us to encode properties of expressions in and of themselves. In this section, we add some vocabulary that allows us to change levels of denotation, i.e. to relate expressions about expressions with the expressions they denote.

The term `(denotation` τ`)` denotes the object denoted by the object denoted by τ. A quotation denotes the quoted expression; the denotation of any other object is $\perp$.

The term `(name` τ`)` denotes the standard name for the object denoted by the term τ. The standard name for an expression τ is `(quote` τ`)`; the standard name for a non-expression is at the discretion of the user. (Note that there are only a countable number of terms in KIF, but there can be models with uncountable cardinality; consequently, it is not always possible for every object in the universe of discourse to have a unique name.)

The final level-crossing vocabulary item is the relation constant `true`. For example, we can say that a sentence of the form `(=> (p ?x) (q ?x))` is true by writing the following sentence.

```
(true '(=> (p ?x) (q ?x)))
```

This may seem of limited utility, since we can just write the sentence denoted by the argument as a sentence in its own right. The advantage of the metanotation becomes clear when we need to quantify over sentences, as in the encoding of axiom schemas. For example, we can say that every sentence of the form `(=>` ϕ ϕ`)` is true with the following sentence.

```
(=> (sentence ?p) (true `(=> ,?p ,?p)))
```

Semantically, we would like to say that a sentence of the form `(true '`ϕ`)` is true if and only if the sentence ϕ is true. In other words, for any interpretation and variable assignment, the truth value t_{iv}`((true '`ϕ`))` is the same as the truth value $t_{iv}(\phi)$. In other words, for every truth function t_{iv}, `true` is our language's name for t_{iv}.

Unfortunately, this causes serious problems. Equating a truth function with the meaning it ascribes to `true` quickly leads to paradoxes. The English sentence "This sentence is false." illustrates the paradox. We can write this sentence in KIF as shown below. The sentence, in effect, asserts its own negation.

```
(true (subst (name `(subst (name x) `x `(true ,x)))
             `x
             `(not (true (subst (name x) `x `(not (true ,x)))))))
```

For any truth function t_{iv} that maps `true` to itself, we get a contradiction. If t_{iv} of this sentence is *true*, then by the rules for assignment of the logical operators contained in the sentence, we see that t_{iv} must make the sentence *false*. If t_{iv} assigns the value *false*, then, again by the rules for assignment of the logical operators, we see that it must assign it the value *true*. In either case, we get a contradication.

Fortunately, we can circumvent such paradoxes by slightly modifying the definition of `true`. The treatment here follows that of Kripke, Gilmore, and Perlis. Although the approach is a little complicated, it is nice in that the intuitive interpretation of `true` is in all important cases exactly what we would guess, yet paradoxes are completely avoided.

(<=> (true ϕ) ϕ^*) (9.31)

Given a sentence ϕ, we define ϕ^* to be the sentence obtained from ϕ as follows. If the sentence is logical, then all occurrences of `not` are pushed inside other operators. If the sentence is (not (true τ)), the ϕ^* is (true (listof 'not τ)).

Since the truth of a sentence (true ϕ) is determined by the truth value of ϕ^*, not ϕ, the potential for paradoxes is eliminated. For most sentences, ϕ^* and ϕ are the same. For apparently paradoxical sentences, the two differ and so no contradiction arises. (See Perlis for the description of a model for databases containing this axiom schema.)

Chapter 10
Nonmonotonicity

Many knowlege representation and reasoning systems are capable of drawing conclusions based on the absence of knowedge from a database. This is nonmonotonic reasoning. The addition of new sentences to the database may be cause for the system to retract earlier conclusions.

In some systems, the exact policy for deriving nonmonotonic conclusions is built into the system. In other systems, the policy can be modified by its user, though rarely within the system's knowledge representation language (e.g. by selecting which predicates to circumscribe). Since KIF is a knowledge representation language and not a system, it is necessary to provide means for its user to express his nomonotonic reasoning policy within the language itself.

We use default rules for this purpose. For instance, the following default rule expresses that an object can be assumed to fly if this object is known to be a bird and it is consistent to assume that it flies.

```
(<<= (flies ?x) (bird ?x) (consis (flies ?x)))
```

The use of `consis` is the only source of nonmonotonicity in KIF. Accordingly, a rule without justifications will be called *monotonic*. This particularly simple case will be discussed first.

§10.1 Monotonic Rules

A *monotonic rule* is an expression of the following form or its reverse (using `=>>`, where ϕ, $\phi_1, \ldots, \phi_n$ are sentences.

$$(\texttt{<<=}\ \phi\ \phi_1\ \ldots\ \phi_n),$$

Such an expression should be distinguished from an implication like the following.

$$(\texttt{<=}\ \phi\ \phi_1\ \ldots\ \phi_n)$$

Athough sentences can be monotonic rules, monotonic rules are not sentences; they are similar to *inference rules*, familiar from elementary logic. If, for instance, Δ consists of some sentences Δ_0 and one rule (`<<=` ψ ϕ), where ϕ and ψ are sentences without free variables, then the set of sentences entailed by Δ is the smallest set of sentences which (i) contains Δ_0, (ii) is closed under logical entailment, and (iii) contains ψ provided that it contains ϕ. It is not generally true that this set contains the implication (`<=` ψ ϕ).

The rationale for using monotonic rules in knowledge representation, instead of implications, is twofold. On the one hand, the "directed" character of rules can simplify the task of developing efficient inference procedures. On the other hand, in some cases, replacing `<<=` by `<=` would be semantically unacceptable. For instance, the rules

```
(<<= (status-known ?x) (citizen ?x))
```

```
(<<= (status-known ?x) (not (citizen ?x)))
```

allow us to infer `(status-known Joe)` only if one of the sentences

```
(citizen Joe),  (not (citizen Joe))
```

can be inferred. Replacing the rules by implications would make `(status-known ?x)` identically true.

§10.2 Logic Programs

A pure Prolog rule

$$\phi\texttt{:-}\phi_1, \ldots, \phi_n$$

where $\phi, \phi_1, \ldots, \phi_n$ are atoms, can be viewed as a syntactic variant of the monotonic rule

$$(\texttt{<<=}\ \ \phi\ \ \phi_1\ \ \ldots\ \ \phi_n)$$

except for two important details. First, the declarative semantics of Prolog applies the *unique names assumption* to its ground terms. If, for example, the program contains no function constants, then this assumption can be expressed by the sentences

$$(\texttt{not}\ \ (=\ \sigma_1\ \ \sigma_2))$$

for all distinct object constants σ_1, σ_2 in the language of the program. Second, this semantics applies the *closed world assumption* to each relation. For a relation constant σ, this assumption can be expressed by the following rule.

$$(\texttt{<<=}\ (\texttt{not}\ (\sigma\ \texttt{@l}))\ (\texttt{consis}\ (\texttt{not}\ (\sigma\ \texttt{@l}))))$$

A pure Prolog program can be translated into KIF by appending to it (i) the sentences expressing the unique names assumption, and (ii) the default rules expressing the closed world assumption.

This method can be easily extended to programs with negation as failure. A negative subgoal `not` ϕ is represented in KIF by the premise `(consis (not` ϕ`))`. (Adding `consis` is necessary because, in KIF, `not` represents classical negation, rather than negation as failure.)

§10.3 Circumscribing Abnormality

Extending a set of sentences by the closed world assumption for some relation constant σ, expressed by a default rule as shown above, has the same effect as circumscribing σ (with all object, function and relation constants varied). In particular, circumscribing abnormality can be expressed by the default rule

```
(<<= (not (ab ?aspect ?x)) (consis (not (ab ?aspect ?x))))
```

Consider, for instance, the nonmonotonic database that contains, in addition to this standard default, two facts.

```
(bird tweety)

(<= (flies ?x) (bird ?x) (not (ab aspect1 ?x)))
```

Birds fly unless they are abnormal in `aspect1`). This database nonmonotonically entails the conclusion that everything is *not* abnormal, including `tweety`:

```
(not (ab ?x))
```

From this, we can conclude that `tweety` flies.

Suppose, on the other hand, that our database includes also the fact that `tweety` is abnormal in `aspect1`:

```
(ab aspect1 Tweety)
```

In this case, we can no longer infer that `tweety` is not abnormal, and, therefore, we cannot conclude that `tweety` is a flier. Note, however, that we have *not* asserted that `tweety` cannot fly; we have only prevented the default rule from taking effect in this case.

Chapter 11
Definitions

KIF includes a set of definition operators for declaring the category and defining axioms (e.g. "Triangles have 3 sides.") for constants. Such *analytic definitions* are intended for use in specifying representation and domain ontologies, and are in contrast to metalinguistic *substitutional definitions* that specify new object level syntax in a macro-like fashion.* KIF definitions can be *complete* in that they specify an expression that is equivalent to the constant, or *partial* in that they specify a defining axiom that restricts the possible denotations of the constant. Partial definitions can be either *unrestricted* or *conservative* extensions to the language. Conservative definitions are restricted in that adding the defining axioms they specify to any given collection of sentences not containing the constant being defined does not logically entail any additional sentences not containing the constant being defined. [Enderton 72].

An analytic definition associates with the constant being defined a *defining axiom.* Intuitively, the meaning of a definition is that its defining axiom is true and that its defining axiom is an *analytic truth.* Analytic truths are considered to be those sentences that are logically entailed from defining axioms. For example, term subsumption in the KL-ONE family of representation languages is an analytic truth in that it is determined solely on the basis of term definitions. The notions of defining axiom and analytic truth are formally defined as follows.

Given a knowledge base Δ, the sentence `(defining-axiom '`σ` '`ϕ`)` means that there is in Δ an analytic definition of constant σ which specifies sentence ϕ as a defining axiom of constant σ. Moreover, defining axioms are true. That is, the following axiom schema holds:

`(=> (defining-axiom '`σ` '`ϕ`) `ϕ`)`

Given a knowledge base Δ, the sentence `(analytic-truth '`ϕ`)` means that the sentence ϕ is logically entailed by the defining axioms of the definitions in knowledge base Δ.

§11.1 Complete Definitions

Complete definitions specify an equivalent term or sentence for the constant being defined as described below. If a constant has a complete definition in a knowledge base, then no other definition for that constant may occur in the knowledge base. Complete definitions are guaranteed to be conservative extensions of the language.

The following table shows the defining axiom specified by each form of complete definition:

* KIF 3.0 does not provide facilities for substitutional definitions. Consideration is being given to including them in later versions of the language.

Definition	Defining Axiom
`(defobject` σ `:=` τ`)`	`(=` σ τ`)`
`(deffunction` π `(`ν_1 `...` ν_n `[`ω`])` `:=` τ`)`	`(=` π `(lambda (`ν_1 `...` ν_n `[`ω`])` τ`))`
`(defrelation` ρ `(`ν_1 `...` ν_n `[`ω`])` `:=` ϕ`)`	`(=` ρ `(kappa (`ν_1 `...` ν_n `[`ω`])` ϕ`))`

Object constants are defined using the `defobject` operator. In the complete definition of an object constant, the first argument, σ, is the constant being defined, and the argument, τ, following the `:=` keyword, is a term. For example, the following definition defines the constant `origin` to be the list (0,0,0).

```
(defobject origin := (listof 0 0 0))
```

The defining axiom specified by this definition of `origin` is:

```
(= origin (listof 0 0 0))
```

Function constants are defined using the `deffunction` operator. In the complete definition of a function constant, the first argument, π, is the constant being defined, the second argument is a list of individual variables with an optional final sequence variable specifying the arguments of the function, and the argument, τ, following the `:=` keyword is a term. For example, the following definition defines the function `paternal-grandfather` in terms of the `father` function.

```
(deffunction paternal-grandfather (?x) := (father (father ?x)))
```

The defining axiom specified by this definition of `paternal-grandfather` is:

```
(= paternal-grandfather (lambda (?x) (father (father ?x))))
```

Relation constants are defined using the `defrelation` operator. In the complete definition of a relation constant, the first argument, ρ, is the constant being defined, the second argument is a list of individual variables with an optional final sequence variable specifying the arguments of the relation; and the argument, ϕ, following the `:=` keyword, is a sentence. For example, the following sentence defines a `bachelor` to be an unmarried man.

```
(defrelation bachelor (?x) := (and (man ?x) (not (married ?x))))
```

The defining axiom specified by this definition of `bachelor` is:

```
(= bachelor (kappa (?x) (and (man ?x) (not (married ?x)))))
```

§11.2 Partial Definitions

A constant can have multiple partial definitions, each of which restricts the possible denotations of the constant. All the definitions of a constant must declare the constant to be the same category; i.e., they must all use the same operator – `defobject`, `deffunction`, or `defrelation`. The defining axioms specified by partial definitions can be either unrestricted or optionally required to be conservative extensions to the language.

Unrestricted Partial Definitions

Unrestricted partial definitions can specify any sentence as a defining axiom, as described below. The following table shows the defining axiom specified by each form of unrestricted partial definition:

Definition	Defining Axiom
`(defobject` σ ϕ_1 ... ϕ_n`)`	`(and` ϕ_1 ... ϕ_n`)`
`(deffunction` π ϕ_1 ... ϕ_n`)`	`(and` ϕ_1 ... ϕ_n`)`
`(defrelation` ρ ϕ_1 ... ϕ_n`)`	`(and` ϕ_1 ... ϕ_n`)`
`(defrelation` ρ `(`ν_1 ... ν_n`)` `:=>` ϕ_1 `:axiom` ϕ_2`)`	`(and (=> (member ?x` ρ`) (= (length ?x) n))` `(=> (`ρ ν_1 ... ν_n`)` ϕ_1`)` ϕ_2`)`
`(defrelation` ρ `(`ν_1 ... ν_n ω`)` `:=>` ϕ_1 `:axiom` ϕ_2`)`	`(and (=> (member ?x` ρ`) (>= (length ?x) n))` `(=> (`ρ ν_1 ... ν_n ω`)` ϕ_1`)` ϕ_2`)`

In an unrestricted partial definition of an object constant, the first argument, σ, is the constant being defined, and the remaining arguments, ϕ_1 ... ϕ_n, are sentences. For example, the following definition defines the constant `id` to be a left and right identity for the binary function `f`.

```
(defobject id (= (f ?x id) ?x) (= (f id ?x) ?x))
```

The defining axiom specified by this definition of `id` (which is just the

conjunction of the second and third arguments in the definition) is unrestricted in that it may contradict other partial definitions of `id` and `f` may not have a left and right identity.

In an unrestricted partial definition of a function constant, the first argument, π, is the constant being defined and the remaining arguments, ϕ_1 ... ϕ_n, are sentences. For example, the following definition defines `f` to be a function which has a value that is greater than 1 for all numbers.

```
(deffunction f (=> (number ?y) (> (f ?y) 1)))
```

The defining axiom specified by this definition of `f` is just the implication that is the second argument in the definition.

There are two basic forms of unrestricted partial definitions for relations. Both forms allow inclusion of an arbitrary sentence to be a defining axiom for the constant being defined. The second form additionally provides for the specification of necessary conditions for the relation to hold. The second form has two variants, depending on whether a sequence variable is included in the function's argument list.

In the first form of unrestricted partial definition of a relation constant, the first argument, ρ, is the constant being defined and the remaining arguments, ϕ_1 ... ϕ_n, are sentences. For example, the following definition defines `R` to be a relation that holds for all single arguments that are positive numbers.

```
(defrelation R (=> (> ?z 0) (R ?z)))
```

The defining axiom specified by this definition of `R` is just the implication that is the second argument in the definition.

In the second form of unrestricted partial definition of a relation constant, the first argument, ρ, is the constant being defined, the second argument is a list of individual variables specifying the arguments of the relation, and the arguments, ϕ_1 and ϕ_2, following the `:=>` and `:axiom` keywords, are sentences. The form has two variants, depending on whether the argument list includes a sequence variable. The following is an example of this form of definition in which `above` is defined to be a binary transitive relation that holds only for "located objects".

```
(defrelation above (?b1 ?b2)
     :=> (and (located-object ?b1) (located-object ?b2))
     :axiom (transitive above))
```

The defining axiom specified by this definition of `above` is:

```
(and (=> (member ?x above) (= (length ?x) 2))
     (=> (above ?b1 ?b2)
         (and (located-object ?b1) (located-object ?b2)))
     (transitive above))
```

Conservative Partial Definitions

Conservative partial definitions specify defining axioms that are conservative extensions of the language. A defining axiom is a conservative extension if adding it to any given collection of sentences not containing the constant being defined does not logically entail any additional sentences not containing the constant being defined. The defining axioms specified by complete definitions and the defining axioms produced directly from some forms of partial definitions are necessarily conservative extensions. However, the arbitrary sentences that can be included in partial definitions are not in general conservative extensions of the language and therefore must be transformed into a conditional form of defining axiom that is guaranteed to be conservative. If a knowledge base contains conservative partial definitions containing arbitrary sentences for a given constant, then those definitions specify a single *conditional defining axiom* for that constant as described below.

The following table shows the defining axiom(s) specified by each form of conservative partial definition:

Definition	Defining Axiom
(defobject σ)	(objconst (quote σ))
(defobject σ :conservative-axiom ϕ)	*The conditional defining axiom for* σ
(deffunction π)	(funconst (quote π))
(deffunction π :conservative-axiom ϕ)	*The conditional defining axiom for* π
(defrelation ρ)	(relconst (quote ρ))
(defrelation ρ :conservative-axiom ϕ)	*The conditional defining axiom for* ρ
(defrelation ρ (ν_1 ... ν_n) :=> ϕ_1)	(and (=> (member ?x ρ) (= (length ?x) n)) (=> (ρ ν_1 ... ν_n) ϕ_1))
(defrelation ρ (ν_1 ... ν_n ω) :=> ϕ_1)	(and (=> (member ?x ρ) (>= (length ?x) n)) (=> (ρ ν_1 ... ν_n ω) ϕ_1))
(defrelation ρ (ν_1 ... ν_n) :=> ϕ_1 :conservative-axiom ϕ_2)	(and (=> (member ?x ρ) (= (length ?x) n)) (=> (ρ ν_1 ... ν_n) ϕ_1)) *The conditional defining axiom for* ρ
(defrelation ρ (ν_1 ... ν_n ω) :=> ϕ_1 :conservative-axiom ϕ_2)	(and (=> (member ?x ρ) (>= (length ?x) n)) (=> (ρ ν_1 ... ν_n ω) ϕ_1)) *The conditional defining axiom for* ρ

There are two forms of conservative partial definitions for objects. In the first form, the argument, σ, is the constant being defined, and the definition simply declares that the constant denotes an object. In the second form, the first argument, σ, is the constant being defined, and the argument, ϕ, following the `:conservative-axiom` keyword is a sentence. The second form of definition provides a sentence to be included in the conditional defining axiom for σ, as described below.

There are two forms of conservative partial definitions for functions. In the first form, the argument, π, is the constant being defined, and the definition simply declares that the constant denotes a function. In the second form, the first argument, π, is the constant being defined and the argument, ϕ, following the `:conservative-axiom` keyword is a sentence. The second form of definition provides a sentence to be included in the conditional defining axiom for π, as described below.

There are three basic forms of conservative partial definitions for relations. In the first form, the argument, ρ, is the constant being defined, and the definition simply declares that the constant denotes a relation. In the second form, the first argument, ρ, is the constant being defined and the argument, ϕ, following the `:conservative-axiom` keyword is a sentence. The second form of definition provides a sentence to be included in the conditional defining axiom for ρ as described below.

The third form of conservative partial definition of a relation constant provides for the specification of necessary conditions for the relation to hold and optionally provides an arbitrary sentence to be included in the constant's conditional defining axiom. The third form has four variants, depending on whether the optional sentence is included and whether a sequence variable is included in the function's argument list.

In the third form of conservative partial definition of a relation constant, the first argument, ρ, is the constant being defined; the second argument is a list of individual variables with an optional final sequence variable, ω, specifying the arguments of the relation; ϕ_1, in the keyword-argument pair, `:=>` ϕ_1, is a sentence; and ϕ_2, in the optional final keyword-argument pair, `:conservative-axiom` ϕ_2, is a sentence. For example, the following definition defines a person to necessarily be a mammal.

```
(defrelation person (?x) :=> (mammal ?x))
```

The defining axiom produced by this definition of `person` is:

```
(and (=> (member ?x person) (= (length ?x) 1))
     (=> (person ?x) (mammal ?x)))
```

The sentences following the keyword `:conservative-axiom` in all of the partial definitions for a given constant are used to form a single conservative defining axiom for that constant. The defining axiom essentially states that if an entity exists in the domain of discourse having all the properties ascribed to the constant by its definitions, then the constant denotes such an entity and the sentences in the constant's definitions following the keyword `:conservative-axiom` are true. That defining axiom is formed as follows.

For a given knowledge base Δ and a given constant σ, let $\phi_1,\ldots,\phi_i$ be the sentences following the keyword `:conservative-axiom` in partial definitions of σ in Δ, and $\phi_{i+1},\ldots,\phi_n$ be the defining axioms otherwise specified in partial definitions of σ in Δ. The sentences $\phi_1,\ldots,\phi_i$ specify the following conservative defining axiom:

`(=> (exists ?x` $\phi_{1(\sigma\rightarrow ?x)}$ ... $\phi_{n(\sigma\rightarrow ?x)}$`)`
`(and` ϕ_1 ... ϕ_i`))`,

where ?x is an individual variable that does not occur in any ϕ_j, and for each `j = 1,...,n`, $\phi_{j(\sigma\rightarrow ?x)}$ is ϕ_j with the following substitutions:

- Each occurrence of σ as a term is replaced by `?x`.
- Each occurrence of `(`σ `<args>)` as a function term is replaced by `(value ?x <args>)`.
- Each occurrence of `(`σ `<args>)` as a relational sentence is replaced by `(holds ?x <args>)`.

This form of defining axiom cannot introduce an inconsistency into a knowledge base since any inconsistency will occur in the antecedent of the implication, thus making the antecedent false and blocking the entailment of the consequent. Also, this form of defining axiom cannot introduce a new domain fact about other constants (e.g., `(color Clyde` grey)), since such a domain fact will occur in the antecedent of the defining axiom and will therefore block the implication of the consequent if it is not already true.

Note that, in general, a constant can have infinitely many partial definitions (meta-linguistically specified by a definition schema). However, if any of the partial definitions of a constant contain a sentence following the keyword `:conservative-axiom`, then the constant must have only a finite number of definitions. Otherwise, the conditional defining axiom for that constant would be an infinitely long sentence, which is not allowed in KIF.

As an example of conservative partial definitions containing arbitrary sentences, consider the following conservative version of the definition given above of `id`, a left and right identity for `f`.

```
(defobject id :conservative-axiom (= (f ?x id) ?x))

(defobject id :conservative-axiom (= (f id ?x) ?x))
```

Assuming there are no other definitions of `id` in the knowledge base, these two partial definitions produce a single defining axiom for `id` as follows:

```
(=> (exists ?y (and (= (f ?x ?y) ?x) (= (f ?y ?x) ?x)))
    (and (= (f ?x id) ?x) (= (f id ?x) ?x)))
```

This axiom states that if there exists a left and right identity for `f`, then `id` is that identity.

The following table summarizes all the forms of KIF definitions and the defining axioms specified by each.

Definition	Defining Axiom
(defobject σ := τ)	(= σ τ)
(defobject σ ϕ_1 ... ϕ_n)	(and ϕ_1 ... ϕ_n)
(defobject σ)	(objconst (quote σ))
(defobject σ :conservative-axiom ϕ)	*The conditional defining axiom for* σ
(deffunction π (ν_1 ... ν_n [ω]) := τ)	(= π (lambda (ν_1 ... ν_n [ω]) τ))
(deffunction π ϕ_1 ... ϕ_n)	(and ϕ_1 ... ϕ_n)
(deffunction π)	(funconst (quote π))
(deffunction π :conservative-axiom ϕ)	*The conditional defining axiom for* π
(defrelation ρ (ν_1 ... ν_n [ω]) := ϕ)	(= ρ (kappa (ν_1 ... ν_n [ω]) ϕ))
(defrelation ρ ϕ_1 ... ϕ_n)	(and ϕ_1 ... ϕ_n)
(defrelation ρ)	(relconst (quote ρ))
(defrelation ρ :conservative-axiom ϕ)	*The conditional defining axiom for* ρ
(defrelation ρ (ν_1 ... ν_n) :=> ϕ_1 [:axiom ϕ_2])	(and (=> (member ?x ρ) (= (length ?x) n)) (=> (ρ ν_1 ... ν_n) ϕ_1)) [ϕ_2])
(defrelation ρ (ν_1 ... ν_n ω) :=> ϕ_1 [:axiom ϕ_2])	(and (=> (member ?x ρ) (>= (length ?x) n)) (=> (ρ ν_1 ... ν_n ω) ϕ_1)) [ϕ_2])
(defrelation ρ (ν_1 ... ν_n) :=> ϕ_1 [:conservative-axiom ϕ_2])	(and (=> (member ?x ρ) (= (length ?x) n)) (=> (ρ ν_1 ... ν_n) ϕ_1)) [*The conditional defining axiom for* ρ]
(defrelation ρ (ν_1 ... ν_n ω) :=> ϕ_1 [:conservative-axiom ϕ_2])	(and (=> (member ?x ρ) (>= (length ?x) n)) (=> (ρ ν_1 ... ν_n ω) ϕ_1)) [*The conditional defining axiom for* ρ]

Chapter A
Abstract Algebra

This appendix contains an ontology for the basic concepts in abstract algebra. The first section gives properties of binary functions. The second section does the same for binary relations. In the third section, these properties are used in defining the a variety of common algebraic structures.

§A.1 Binary Operations

```
(defrelation binop (?f ?s) :=
  (and (binary-function ?f)
       (subset (universe ?f) ?s)))                                  (A.1)

(defrelation associative (?f ?s) :=
  (forall (?x ?y ?z)
    (=> (member ?x ?s) (member ?y ?s) (member ?z ?s)
        (= (value ?f ?x (value ?f ?y ?z))
           (value ?f (value ?f ?x ?y) ?z)))))                       (A.2)

(defrelation commutative (?f ?s) :=
  (forall (?x ?y)
    (=> (member ?x ?s) (member ?y ?s)
        (= (value ?f ?x ?y) (value ?f ?y ?x)))))                    (A.3)

(defrelation invertible (?f ?o ?s) :=
  (forall (?x)
    (=> (memberp ?x ?s)
        (exists (?y)
          (and (member ?y ?s)
               (= (value ?x ?y) ?o) (= (value ?y ?x) ?o))))))       (A.4)

(defrelation distributes (?f ?g ?s) :=
  (and (binop ?f ?s) (binop ?g ?s)
       (forall (?x ?y ?z)
         (=> (member ?x ?s) (member ?y ?s) (member ?z ?s)
             (= (value ?f (value ?g ?x ?y) ?z)
                (value ?g (value ?f ?x ?z)
                          (value ?f ?y ?z)))))))                    (A.5)
```

§A.2 Binary Relations

```
(defrelation binrel (?r ?s) :=
  (and (binary-relation ?r)
```

```
        (subset (universe ?r) ?s)))                                   (A.6)

  (defrelation reflexive (?r ?s) :=
    (and (binrel ?r ?s)
         (forall ?x (=> (member ?x ?s)
                        (holds ?r ?x ?x)))))                          (A.7)

  (defrelation irreflexive (?r ?s) :=
    (and (binrel ?r ?s)
         (forall (?x)
                 (=> (member ?x ?s) (not (holds ?r ?x ?x))))))        (A.8)

  (defrelation symmetric (?r ?s) :=
    (and (binrel ?r ?s)
         (forall (?x ?y) (=> (holds ?r ?x ?y) (holds ?r ?y ?x)))))    (A.9)

  (defrelation asymmetric (?r ?s) :=
    (and (binrel ?r ?s)
         (forall (?x ?y) (=> (holds ?r ?x ?y))
                             (not (holds ?r ?y ?x))))))               (A.10)

  (defrelation antisymmetric (?r ?s) :=
    (and (binrel ?r ?s)
         (forall (?x ?y)
           (=> (holds ?r ?x ?y) (holds ?r ?y ?x) (= ?x ?y)))))        (A.11)

  (defrelation trichotomizes (?r ?s) :=
    (and (binrel ?r ?s)
         (forall (?x ?y)
           (=> (member ?x ?s) (member ?y ?s)
               (or (holds ?r ?x ?y)
                   (= ?x ?y)
                   (holds ?r ?y ?x))))))                              (A.12)

  (defrelation transitive (?r ?s) :=
    (and (binrel ?r ?s)
         (forall (?x ?y ?z)
           (=> (holds ?r ?x ?y) (holds ?r ?y ?z)
               (holds ?r ?x ?z)))))                                   (A.13)
```

§A.3 Algebraic Structures

```
  (defrelation semigroup (?s ?f ?o) :=
    (and (binop ?f ?s)
```

```
      (associative ?f ?s)
      (identity ?o ?f ?s)))                                    (A.14)

(defrelation abelian-semigroup (?s ?f ?o) :=
  (and (semigroup ?s ?f ?o)
       (commutative ?f ?s)))                                   (A.15)

(defrelation group (?s ?f ?o) :=
  (and (binop ?f ?s)
       (associative ?f ?s)
       (identity ?o ?f ?s)
       (invertible ?f ?o ?s)))                                 (A.16)

(defrelation abelian-group (?s ?f ?o) :=
  (and (group ?s ?f ?o)
       (commutative ?f ?s)))                                   (A.17)

(defrelation ring (?s ?f ?o ?g ?i) :=
  (and (abelian-group ?s ?f ?o)
       (semigroup ?s ?g ?i)
       (distributes ?g ?f ?s)))                                (A.18)

(defrelation commutative-ring (?s ?f ?o ?g ?i) :=
  (and (abelian-group ?s ?f ?o)
       (abelian-semigroup ?s ?g ?i)
       (distributes ?g ?f ?s)))                                (A.19)

(defrelation integral-domain (?s ?f ?o ?g ?i) :=
  (and (commutative-ring ?s ?f ?o ?g ?i)
       (operation ?g (difference ?s (setof ?o)))))             (A.20)

(defrelation division-ring (?s ?f ?o ?g ?i) :=
  (and (ring ?s ?f ?o ?g ?i)
       (binop ?g (difference ?s (setof ?o)))
       (invertible ?g (difference ?s (setof ?o)))))            (A.21)

(defrelation field (?s ?f ?o ?g ?i) :=
  (and (division-ring ?s ?f ?o ?g ?i)
       (commutative ?f ?s)))                                   (A.22)

(defrelation partial-order (?s ?r) :=
  (and (irreflexive ?r ?s)
       (asymmetric ?r ?s)
       (transitive ?r ?s)))                                    (A.23)
```

```
(defrelation linear-order (?s ?r) :=
  (and (irreflexive ?r ?s)
       (asymmetric ?r ?s)
       (transitive ?r ?s)
       (trichotomizes ?r ?s)))                                   (A.24)
```