A database interface based on Montague's approach to the interpretation of natural language

R. A. Frost and W. S. Saba
School of Computer Science, University of Windsor, Windsor, Ontario N9B 3P4, Canada.

(Received 30 March 1988 and in revised form 27 June 1989)

In this paper we describe a database interface that is loosely based upon some of the concepts proposed by Richard Montague in his approach to the interpretation of natural language. The system is implemented as an executable attribute grammar specified in a higher order, lazy, pure functional programming language. The attribute grammar formalism provides a simple means of implementing Montague's notion of "semantic rule to syntactic rule correspondence" and the higher order functional language in which the attribute grammar is constructed provides an appropriate vehicle for implementing Montague's higher order semantics. The purpose of the paper is two-fold: (i) to demonstrate that many of Montague's ideas can be used to advantage in creating natural language interfaces to databases, and (ii) to introduce a method for implementing attribute grammars in functional languages that is suitable for investigating both grammars and semantic theories of language.

1. Introduction

Montague (1971) regarded natural languages as formal languages. In particular, he claimed that (i) the objects denoted by phrases of any natural language denote functions in a function space constructed over a set of objects of a few "primitive" types; (ii) for each syntactic category of a natural language there is a corresponding semantic type; and (iii) for each syntactic rule that shows how a complex syntactic construct can be built from simpler constructs, there is a corresponding semantic rule that shows how the meaning of the complex construct can be computed from the meanings of its parts.

The motivation for the work, of which this paper is a part, resulted from two observations:

(i) The higher order, strongly typed nature of Montague's semantics suggests that the relatively new higher order, pure, strongly typed functional programming languages may be appropriate implementation languages for systems based on Montague's approach to the interpretation of natural language.
(ii) The correspondence between syntactic and semantic rules suggests that "attribute grammars" could be used to advantage in systems based on Montague's notions.

In this paper we present a database system, DMSG, that is loosely based on Montague's approach to the interpretation of natural language. DMSG can answer various questions expressed in English with respect to a database containing facts about the solar system. DMSG is implemented in the higher order, strongly typed,
lazy, pure functional programming language Miranda† (Turner, 1985) and is structured around an interpreter that is based on an attribute grammar (Knuth, 1968). The system that we have built is only “loosely” based on Montague’s notions in that (i) we have used only a few of Montague’s ideas, and (ii) we have taken some liberty with those ideas that we have used. However, we hope to show that the combination of Montague’s principles with a higher order, pure functional programming language, such as Miranda, is an area that deserves further exploration.

The paper is structured as follows:

(i) This introduction.
(ii) A brief discussion of some of Montague’s notions and how we have used them.
(iii) A brief discussion of functional programming and the language Miranda.
(iv) A brief introduction to functional attribute grammars.
(v) An example interactive session with DMSG.
(vi) A nearly complete commented listing of the Miranda code for DMSG.

We conclude with some comments on future work.

2. Aspects of Montague’s approach that we have used

Montague believed that natural languages are formal languages whose syntax and semantics can be defined precisely. Montague’s method for assigning an interpretation to a phrase of natural language involves four steps:

(i) The natural language phrase is analysed using a set of syntactic rules. Each syntactic rule contains information specifying how complex expressions of given syntactic categories can be constructed from simpler components of given syntactic categories. More than one rule may be used to analyse a phrase.
(ii) Each syntactic rule has a translation rule associated with it. These translation rules specify the translation of the outputs from the syntactic rules in terms of the inputs to the syntactic rules. The translation rules specify how to translate natural language expressions to expressions in an unambiguous formal language of an intensional logic called IL, i.e. a language that has a well defined model theoretic semantics. The translation of a single natural language expression into possibly many expressions in the unambiguous language is often referred to as “disambiguation”.
(iii) The semantics of IL tells us that each component of the expression that is output from the translation rules denotes a function in a function space constructed over a set of constant functions corresponding to:

—Two objects, “True” and “False”.
—A set of entities.
—A set of “times”.
—A set of “possible worlds”.

† Miranda is a trademark of Research Software Ltd.
(iv) The meaning of the IL expression is obtained by reference to the model theoretic semantics underlying IL.

Montague's intensional logic IL is a typed higher-order modal intensional predicate calculus that includes aspects of lambda calculus. The type theory employed is that of Church (1940). Comprehensive accounts of IL and Montague's work in general can be found in Thomason (1974), and Dowty, Wall & Peters (1981).

Montague indicated quite clearly that the use of IL as an intermediary language is not strictly necessary and serves only to facilitate the process of interpretation. In our approach IL is not used at all. Functions are encoded in Miranda. This is a more primitive language than that of IL. It is more primitive in the sense that it does not have, for example, a built-in, universal quantifier "operator" as IL does. Also we have not incorporated any modal or intensional concepts. Consequently, we are using only a few of Montague's ideas. More importantly, we differ from Montague in that the functions, that we regard as being denoted by words, are closely related to set theory rather than to a calculus of characteristic functions of relations. For example, we regard intransitive verbs as denoting constant functions whose values are sets of entities. Montague regarded intransitive verbs as denoting characteristic functions of unary relations of entities, i.e. functions that take an entity as argument and return a boolean value as result depending on whether or not the entity is in some unary relation. The main advantage of our approach is efficiency; the main disadvantage is some loss of polymorphism. For example, in Montague's approach it is possible to associate a single function with the word "and". In a set theoretic approach, different functions are denoted by "and" depending upon the type of phases being "conjoined". Other than this, the conversion from Montague's calculus of characteristic functions to set theoretic functions is relatively straightforward. The main concepts that we have borrowed from Montague, include:

—The many-to-one correspondence between syntactic categories and semantic types.
—The rule-to-rule correspondence: the notion that there is a semantic rule corresponding to every syntactic/formation rule.
—The notion that the semantic domain includes higher-order functions.
—The notion that some phrases of English can be regarded as denoting functions that have been partially applied. For example, the notion that the phrase "every man" denotes a function that is the result of partially applying the two argument function denoted by "every" to the single argument denoted by "man".
—The notion that termphrases do not denote entities directly. That is, a termphrase such as "John" does not denote an entity but rather a function that is in some way related to a specific entity.

3. The implementation language

The language used in the implementation is the higher-order, lazy, pure functional programming language Miranda. However, before we describe Miranda, we briefly discuss some of the features of pure functional programming.
3.1. PURE FUNCTIONAL PROGRAMMING

The design of any system involves identification of a configuration of available components to perform the tasks defined by the given specification. Irrespective of the type of system being built, the design process is such that:

—The specification defines the required system as returning given values for given inputs within given constraints.
—The available components may themselves be specified as systems.
—The construction strategies used to configure the components may also be regarded as systems that return more complex components from simpler components given as input.

Inherent in any design process is the use of abstraction. Abstraction is the ability to regard systems as black boxes that comply with particular specifications. The advantage of abstraction is that it facilitates the construction of complex systems since the designer can ignore the inner workings of the black boxes developed at each stage. In the domain of computer programming, the advantages of abstraction with respect to available components have been appreciated for some time and this has resulted in the notions of abstract data types, procedures, subroutines, and program packages. Most modern programming languages support these notions. The advantages of regarding construction strategies as black boxes have been appreciated only relatively recently. However, this appreciation is growing and is reflected in a renewed interest in functional programming. In "pure" functional programming, all systems are defined as functions. The functions may be of a different type but have the same status in all other respects. In particular, no distinction is made between first order and higher order functions. A higher order function is a function that takes a function as input and/or returns a function as result. Higher order functions may be regarded as construction strategies that enable new functions to be configured using existing resources. An example of a higher order function is a function "foldr" that can be used to construct list processing functions from binary operators and elements, for example:

\[
\begin{align*}
\text{sumlist} & = \text{foldr} (+) 0 \\
\text{productlist} & = \text{foldr} (*) 1 \\
\text{andlist} & = \text{fold} (&) \text{True} \\
\text{orlist} & = \text{foldr}(\lor) \text{False}
\end{align*}
\]

That is, "sumlist" is obtained by applying the function "foldr" to the binary operator "+" and the element "0". The resulting function takes a list as input and returns the sum of its elements as result, "productlist" returns the product of the elements, "andlist" returns the conjunction, and "orlist" the disjunction.

The construction of a program in a pure functional programming language involves the definition of a function that complies with the given specification in terms of other functions using construction strategies that are themselves defined as functions. The resources available to the programmer comprise a library of functions, both first order and higher order, whose specifications are known.

3.2. INTRODUCTION TO MIRANDA

Miranda is a "pure" functional programming language. It is pure, in the sense that it has no procedural aspects and function application returns a value with no side
effects whatsoever. Miranda is a higher order language in that functions can take functions as arguments and/or return functions as result. Miranda allows partial parameterization: functions of $n$ arguments may be defined and used as functions of one argument whose values are functions of $n - 1$ arguments. Miranda employs a "lazy evaluation" strategy resulting in efficient execution and also enabling infinite objects to be defined and used as arguments to functions. In lazy evaluation, arguments of functions are only evaluated when required and only to the extent required. Lists may be defined in Miranda by enumeration of their elements or by the use of "comprehensions" which are constructed using a syntax based on Zermelo–Fraenkel set notation. Miranda is strongly typed; the type theory used is that of Church (1940). Polymorphic functions can be defined in Miranda enabling the abstraction and construction of general purpose functions. Miranda is well suited to our task for a number of reasons:

—Miranda has a theoretical basis that is very similar to that underlying Montague's approach. Miranda is based on typed combinatory logic and Montague's approach is based on a typed lambda calculus. The type theory in both cases is that of Church. Combinatory logic and lambda calculus are formal systems that were designed for studying certain primitive ways in which functions can be combined to form other functions.

—Abstraction and modularity are supported owing to the fact that functions do not have side effects, and higher order functions can be defined.

—Miranda is sufficiently high level and "declarative" to be regarded as an executable specification language.

4. Functional attribute grammars

Ever since the introduction of definite clause grammars (Pereira & Warren, 1978), Prolog has been a natural choice for experimenting with grammars and semantic theories of language. During the last two years, we have implemented a similar scheme for lazy, pure functional programming languages. We have developed a number of higher order functions that can be used to "glue" together parts of an attribute grammar specification such that the result is an executable interpreter.

Our system of functions enables researchers to address both syntactic and semantic issues methodically and create language interpreters that are highly modular. Construction of an interpreter involves the following tasks:

—Definition of a grammar for the language to be interpreted. The grammar must not be left recursive.

—Identification of relevant attributes for each construct of the language. This involves naming attribute types and defining attribute types in terms of some primitive base types.

—Specification of a dictionary consisting of a number of lists of words paired with their meanings. There is a separate list for each of the basic syntactic categories of the language to be interpreted. The "meaning" of each word is a list of attribute values associated with the word when used in the given syntactic context. A word can appear in more than one list and more than once in a list. The meaning of a word may also be defined in terms of a given phrase of the language by use of a
higher order function called "meaning_of". This function calls the interpreter recursively in order to obtain a list of attributes.

—Specification of an executable attribute grammar for the language to be interpreted. This grammar states, declaratively, how attributes for each construct of the language are synthesised from attributes of its components. The attribute grammar also allows inherited attributes, thereby accommodating context dependencies which can be used, for example, in the resolution of pronouns.

—Definition of the attribute functions, referred to in the attribute grammar, in terms of operations from the calculi on which the semantic theory is founded. In DMSG, functions on attributes are defined in terms of operations from set theory and relational algebra.

—Definition of the database, or of an interface to a database. In DMSG, the database consists of a number of unary and binary relations.

We present our method by way of an example. We give an example interactive session with DMSG and then present a nearly complete listing of the executable specification of DMSG.

5. An example interactive session with DMSG

The purpose of this paper is to show that many of Montague's ideas can be used to advantage in creating natural language interfaces to databases, and to introduce a method for implementing functional attribute grammars that is suitable for investigating both grammars and semantic theories of language.

The grammar and semantic theory that we have used in DMSG have many shortcomings and we do not claim that they are adequate for anything other than as an example. However, they are plausible enough for this purpose as can be seen from the example interactive session below.

In the following, user input is in italic font and the interpreter's response in bold. The answers returned are dependent on the limited database that is built into DMSG. The word "blue" appears twice in the dictionary for adjectives. One entry is for the colour blue, and one for the psychological feeling of blueness. We have taken the liberty of telling DMSG that Hall was depressed. DMSG also believes that no-one has been credited with discovery of any of the planets.

Miranda session

Hello. I can answer some questions posed in a limited subset of English. My knowledge covers the planets, their moons and discoverers. Please end all questions with a question mark. Use (Control-D) to finish.

do all moons spin?
yes
how many people discovered a moon that orbits mars?
one.
who discovered a moon that orbits.mars?
Hall.
did Hall discover anything?
yes.
Hall is a blue man.
The question is ambiguous. The possible answers are
* false.
* true.
Hall is a discoverer?
true.
which moon that orbits mars was discovered by Hall?
phobos, and deimos.
Hall discovered phobos and it is a moon?
true.
Hall discovered phobos and he is a man?
true.
Hall discovered phobos and he is a moon?
false.
phobos orbits mars and it is a planet?
false.
Hall or Kuiper discovered a moon that orbits mars?
true.
which planets spin?
mercury, venus, earth, mars, jupiter, saturn, uranus, neptune, and pluto.
which planets that are red and solid spin?
mars.
which red solid planets spin?
mars.
which red planets that are solid spin?
mars.
which planets that are solid red and atmospheric spin?
mars.
did anyone discover mars?
no.
which discoverer who is a man discovered phobos?
Hall.
which red planet is orbited by a moon that was discovered by Hall?
mars.
did Hall discover a moon that orbits jupiter or mars?
The question is ambiguous. The possible answers are
* yes.
* no.
Hall is the person who discovered phobos?
true.
did Hall discover a moon that orbits mars and a moon?
The question is ambiguous. The possible answers are
* no.
* yes.
what was discovered by Galileo?
io, europa, ganymede, and callisto.
what orbits mars?
phobos, and deimos.
what is orbited by mars?
sol.
which planets are orbited by the moons that were discovered by Kuiper?
uranus, and neptune.
how many red planets that are solid exist?
one.
who discovered a moon that is red and solid?
Galileo.

6. The executable specification of DMSG

We now present a nearly complete commented executable specification of DMSG in the functional language Miranda. The specification is complete except that (i) we have omitted definitions of the higher order library functions that are used to "glue" together the executable specification, and (ii) we have omitted some list entries and some parts of function definitions where we feel that such omission will not impair understanding of the specification. A complete listing of DMSG, which is approximately 1000 lines long without comments, is available in a technical report (Frost & Saba, 1989).

In the following listing, Miranda code appears in bold font and comments in regular font. Readers who are unfamiliar with the functional style of programming are referred to Bird and Wadler (1988) as an introductory text.

---

**BASIC GRAMMAR OF THE QUERY LANGUAGE**

The basic grammar below is given in standard BNF notation with its usual interpretation. This basic grammar is not part of the executable specification and is given only as part of the documentation.

Note that "snoun" stands for "simple noun", "cnoun" for "common noun", "pnoun" for "proper noun", "cla" for "clause", "vb" for "verb", "ph" for "phrase", "pron" for "pronoun", "pass" for "passive", and "det" for "determiner".

```
snouncla ::= cnoun
  | adjs
  | adjs cnoun

relnouncla ::= snouncla relpron joinvbph
  | snouncla

nouncla ::= relnouncla nounjoin nouncla
```

---
DATABASE INTERFACE

**BASIC SEMANTIC TYPES (in addition to Miranda's existing primitive types)**

```plaintext
entity  ==  num
entityset  ==  [entity]
string  ==  [char]
```
**ATTRIBUTE TYPES**

With each of the syntactic constructs we associate a "value" attribute type. For example, "cnouns" have "cnoun_val" attributes which are constructed from objects of type "entityset" using the constructor "CNOUN_VAL". Some constructs will also have "sort" and "number" attributes associated with them. The "poss_subjs" attributes are lists of lists of attributes of possible subjects that could be substituted for pronouns found later in a question. The type definitions below should be read as indicating, for example, that an attribute of type "det_val" can be obtained by applying the constructor "DET_VAL" to a function that takes two entitysets as argument and which returns a boolean value as result.

```
attributes ::= 
SORT string
| NUMBER string
| CNOUN_VAL entityset
| INTRANSVB_VAL entityset
| ADJ_VAL entityset
| DET_VAL (entityset→entityset→bool)
| PNOUN_VAL (entityset→bool)
| TRANSVB_VAL [(entityset→bool)→entityset]
| PASSTRVVB_VAL ((entityset→bool)→entityset)
| RELPRON_VAL (entityset→entityset→entityset)
| NOUNJOIN_VAL (entityset→entityset→entityset)
| VEBPHJOIN_VAL (entityset→entityset→entityset)
| TERMPHJOIN_VAL [(entityset→bool)→(entityset→bool)→(entityset→bool)]
| SNOUNCLA_VAL entityset
| RELNOUNCLA_VAL entityset
| ADJS_VAL entityset
| NOUNCLA_VAL entityset
| DETPH_VAL (entityset→bool)
| TERMPH_VAL (entityset→bool)
| VERBPPh_VAL entityset
| TRANSVBPPh_VAL entityset
| JOINTERMPh_VAL (entityset→bool)
| PREP_VAL (entityset→entityset)
| JOINVBPH_VAl entityset
| SENT_VAl bool
| LINKINGVb可以更好更的实体set
| INDEFPRON_VAL (entityset→bool)
ASSOCIATING STRINGS WITH CONSTRUCTORS

The functions "name" associates strings with attribute constructors. This function is used later to pick particular attribute values from lists of attributes.

```lisp
name (NUMBER x) = "number"
name (SORT x) = "sort"
name (CNOUN_VAL x) = "cnoun_val"
```

etc.

THE DICTIONARY

The dictionary consists of a list for each of the basic syntactic categories. The first list is for common nouns. Each list consists of words paired with sets of attributes that are associated with the word when used in the given syntactic category. For example, the word "man" when used as a common noun is paired with three attributes:

- the first is obtained by applying the constructor "CNOUN_VAL" to an entityset called "set_of_men", the second is obtained by applying the constructor "SORT" to the string "anim", and the third is obtained by applying the constructor "NUMBER" to the string "singlr".

The word "human" is paired with a set of attributes that are obtained by applying the interpreter recursively, through the function "meaning_of" to the phrase "man or woman". In our example, we are not computing SORT and NUMBER attributes of complex phrases from the SORT and NUMBER attributes of their components (although this would be easy to do, it would result in a longer program than is desirable for the purpose of this paper). Consequently,
we specify that these extra attributes should be associated with
the word "human." The conversion from "nouncla_val" attribute
to "cnoun_val" attribute, in the entry for "human", is necessary since the
value associated with the phrase "man or woman," will be of type
"nouncla_val", whereas the appropriate type for "human"
is "cnoun_val".

Note that semantic objects, such as "set_of_men",
are defined later in the database part of the specification.

cnoun_list =
[("man", [CNOUN_VAL set_of_men, SORT "anim", NUMBER "single"]),
("men", [CNOUN_VAL set_of_men, SORT "anim", NUMBER "plural"]),
("thing", [CNOUN_VAL set_of_thing, SORT "inanim", NUMBER "single"]),
("things", [CNOUN_VAL set_of_thing, SORT "inanim", NUMBER "plural"]),
("planets", [CNOUN_VAL set_of_planet, SORT "inanim", NUMBER "plural"]),
("planet", [CNOUN_VAL set_of_planet, SORT "inanim", NUMBER "single"]),
("human", (meaning_of nouncla "man or woman".
(with_extra_atts [SORT "anim", NUMBER "single"]))
(with_conv ["cnoun_val", "nouncla_val"])),

adjective_list =
[("atmospheric", [ADJ_VAL set_of_atmospheric, SORT "inanim"]),
("blue", [ADJ_VAL set_of_blue, SORT "any"]),
("depressed", [ADJ_VAL set_of_depressed, SORT "anim"]),

intransvb_list =
["exist", [INTRANSVB_VAL set_of_exist, NUMBER "plural"]],
("exists", [INTRANSVB_VAL set_of_exist, NUMBER "single"]),
(orbit", (meaning_of verbph "orbit something".
(with_extra_atts [NUMBER "plural"])
(with_conv ["intransvb_val", "verbph_val"]))),

etc.

The semantic objects that are denoted by determiners, such
as "function_denoted_by_a", are defined in the semantics
part of the specification.

det_list =
["the", [DET_VAL function_denoted_by_a, NUMBER "any"]],
("a", [DET_VAL function_denoted_by_a, NUMBER "single"]),
("one", [DET_VAL function_denoted_by_a, NUMBER "single"]),

indefpron_list =
["anyone", (meaning_of detph "a person."
with_no_extra_atts
(with_conv ["indefpron_val", "detph_val"])),
("anything", (meaning_of detph "a thing."
with_no_extra_atts
(with_conv ["indefpron_val", "detph_val"])),

etc.
Proper nouns do not denote entities. Rather, they are related to functions that are defined in terms of particular entities associated with the proper nouns. In DMSG, each proper noun denotes a function that is obtained by applying the higher order function "test_wrt" to some entity. In each case, the result is a function that takes a set as argument and which returns a boolean value depending on whether or not this set contains the entity associated with the proper noun. For example:

\[ \text{test_wrt} \ 12 \ [1, 2, 7, 12, 15] \Rightarrow \text{True} \]

pnoun_list =

\[
(\text{"Bernard"}, \ \text{PNOUN}_\text{VAL} (\text{test_wrt} 55), \text{SORT} \ "\text{anim}")],
(\text{"Bond"}, \ \text{PNOUN}_\text{VAL} (\text{test_wrt} 67), \text{SORT} \ "\text{anim}")],
(\text{"venus"}, \ \text{PNOUN}_\text{VAL} (\text{test_wrt} 10), \text{SORT} \ "\text{anim}")],
\]

etc.

Transitive verbs denote objects that are obtained by applying the function "trans_verb" to an appropriate binary relation. For example, "discover" denotes a function that is obtained by applying the function "trans_verb" to the relation "rel_discover". An explanation of what "trans_verb" does can be found in Frost and Launchbury (1989).

\[
\text{transvb_list} =
(\text{"discover"}, \ \text{TRANSVB}_\text{VAL} (\text{trans_verb rel_discover}))],
(\text{"discovered"}, \ \text{TRANSVB}_\text{VAL} (\text{trans_verb rel_discover}))],
(\text{"orbit"}, \ \text{TRANSVB}_\text{VAL} (\text{trans_verb rel_orbit}))],
\]

etc.

Passive verbs denote objects that are obtained by applying the function "passtr_verb" to an appropriate binary relation. The advantage of this approach is that an active and a passive transitive verb can both be defined in terms of a single binary relation.

\[
\text{passtrvb_list} =
(\text{"discovered"}, \ \text{PASSTRVB}_\text{VAL} (\text{passtr_verb rel_discover}))],
(\text{"orbited"}, \ \text{PASSTRVB}_\text{VAL} (\text{passtr_verb rel_orbit}))]
\]

DMSG does not accommodate tense at present. Hence, linking verbs have no effect and are regarded as denoting the identity function.

\[
\text{linkingvb_list} =
(\text{"is"}, \ \text{LINKINGVB}_\text{VAL} \ \text{id}]),
(\text{"was"}, \ \text{LINKINGVB}_\text{VAL} \ \text{id}]),
(\text{"are"}, \ \text{LINKINGVB}_\text{VAL} \ \text{id}]),
\]

etc.
Relative pronouns and conjoiners of verbphrases and nounclauses denote appropriate functions from set theory. For example, the verbphrase conjoiner “and” in DMSG denotes the function of set intersection.

relpron_list = 
[(“that”, [RELPRON_VAL intersect]),
 (“who”, [RELPRON_VAL intersect]),
 (“which”, [RELPRON_VAL intersect])]

verbphjoin_list = 
[(“and”, [VBPHJOIN_VAL intersect]),
 (“or”, [VBPHJOIN_VAL union])]

nounjoin_list = 
[(“and”, [NOUNJOIN_VAL intersect]),
 (“or”, [NOUNJOIN_VAL union])]

prep_list = 
[  (“by”, [PREP_VAL id]) ]

Conjoiners of termphrases denote functions that are defined in the semantics part of the specification.

termphjoin_list = 
[(“and”, [TERMPHJOIN_VAL termph_and]),
 (“or”, [TERMPHJOIN_VAL termph_or])]

terminator_list = 
[  (“.”, [DOT_VAL []]),
   (“?”, [QM_VAL []]) ]

pronoun_list = 
[(“he”, [SORT “anim”, NUMBER “singlr”]),
 (“it”, [SORT “inanim”, NUMBER “singlr”]) ]

sentjoin_list = 
[  (“and”, [SENTJOIN_VAL sand]) ]

question_type1 = 
[  (“does”, [QUESTI_VAL yesno]),
   (“did”, [QUESTI_VAL yesno]),
   (“do”, [QUESTI_VAL yesno]) ]

question_type2 = 
[  (“what”, [QUEST2_VAL whewhatq, SORT “any”]),
   (“who”, [QUEST2_VAL whewhatq, SORT “anim”]) ]
question_type3 =
    [ ("which", [QUEST3_VAL whichq]) ]

question_type4a=
    [ ("how", [QUEST4_VAL howmanyq]) ]

question_type4b=
    [ ("many", [QUEST4_VAL howmanyq]) ]

BASIC INTERPRETERS

Basic interpreters for words are obtained by applying
the higher order function “make_interpreter_from” to each
of the lists from the dictionary.
As an example of what a basic interpreter does, consider the
following application of the interpreter “cnoun”:

cnoun [] (words “planet xxxx ...”) ⇒
    [[CNOUN_VAL [9,10,11,12,13,14,15,16,17],
      SORT "inanim",NUMBER "singlr"],
     [“xxxx”, “ ’”, “.”, “.”, “.”, “.”]]

The interpreter “cnoun” translates the word “planet”
to a list of attributes, and pairs this with the tail of
the input string. There is only one entry for “planet”, and therefore only
one entry in the result returned by “cnoun”.

pnoun = make_interpreter_from pnoun_list
cnoun = make_interpreter_from cnoun_list
adj = make_interpreter_from adjective_list
etc.

THE ATTRIBUTE GRAMMAR

The attribute grammar comprises a number of executable
specifications, one for each non-basic syntactic
category of the query language. Each specification
indicates how attributes of the syntactic construct
can be computed from the attributes of its components.
For example, a simple noun clause can consist of a common noun
or else it can consist of a set of adjectives (this enables
constructs such as “which planets that are solid, red and
atmospheric spin?”), or else it can consist of a set of
adjectives followed by a common noun. If the simple noun
clause is of the latter type, its “snouncla_val” attribute
is computed by applying the function “intrsctl” (defined
in the semantics section) to the “cnoun_val” attribute of the
first part, i.e. the adjectives, and the “cnoun_val” of the second part,
i.e. the common noun.
The functions “conv1” and “conv15” are used to convert
attributes to be of an appropriate type to be returned
by the “snouncla” interpreter.
The specification of the “snouncla” interpreter
does not involve any inherited attributes.
However it does involve an example of the use of a failure
rule. If the sorts of the components of a common noun clause
are incompatible, then the parse fails. For example:
interpret “Hall is a red man”. ⇒
    “false”
interpret “Hall is an electric man”. ⇒
    “I do not understand”

This is not the best way to handle “sort clashes” (a more
appropriate approach would be to return an error message).
However, it serves to illustrate the use of failure rules
in the attribute grammar.

\[
\text{snouncla} = \begin{cases}
\text{consists of [cnoun]} \\
\text{with att syn rules}
\end{cases}
\begin{align*}
\left[\left(\text{"snouncla_val"}, \"is\", \text{convl}, \left[\left(\text{"cnoun_val"}, \text{fst}_\text{part}\right)\right]\right), \\
\left(\text{"sort"}, \"is\", \text{same}, \left[\left(\text{"sort"}, \text{fst}_\text{part}\right)\right]\right)\right] \\
\text{with no inh att calc rules} \\
\text{with no failure rules} \\
\text{orelse} \\
\text{consists of [adjs]} \\
\text{with att syn rules}
\end{align*}
\begin{align*}
\left[\left(\text{"snouncla_val"}, \"is\", \text{conv15}, \left[\left(\text{"adjs_val"}, \text{fst}_\text{part}\right)\right]\right), \\
\left(\text{"sort"}, \"is\", \text{same}, \left[\left(\text{"sort"}, \text{fst}_\text{part}\right)\right]\right)\right] \\
\text{with no inh att calc rules} \\
\text{with no failure rules} \\
\text{orelse} \\
\text{consists of [adjs,cnoun]} \\
\text{with att syn rules}
\end{align*}
\begin{align*}
\left[\left(\text{"snouncla_val"}, \"is\", \text{intrsctl}, \\
\left[\left(\text{"adjs_val"}, \text{fst}_\text{part}\right), \left(\text{"cnoun_val"}, \text{snd}_\text{part}\right)\right]\right), \\
\left(\text{"sort"}, \"is\", \text{same}, \left[\left(\text{"sort"}, \text{fst}_\text{part}\right)\right]\right)\right] \\
\text{with no inh att calc rules} \\
\text{with failure rules}
\end{align*}
\begin{align*}
\left[\left(\text{check for clash}, \left[\left(\text{"sort"}, \text{fst}_\text{part}\right), \left(\text{"sort"}, \text{snd}_\text{part}\right)\right]\right)\right]
\end{align*}
\]
Each interpreter will return a list of all successful parses. For example, consider the following:

relnouncla [] (words “planet that is orbited by phobos xxxx”.) \[\Rightarrow\]

\[([\text{RELNOUNCLA VAL} [9,10,11,12,13,14,15,16,17]],\]

\[\text{["that","is","orbited","by","phobos","xxxx","."]},\]

\[([\text{RELNOUNCLA_VAL} [12]],\text{["xxxx","."}]),\]

That is, the interpreter “relnouncla” when applied to the
input string “planet that is orbited by phobos xxxx.” returns two
results, one for the “snouncla” “planet”, and one for the
“snouncla, relpron, joinvbph” “planet that is orbited by phobos”.

\[\text{relnouncla = consists_of [snouncla, relpron, joinvbph]}\]
\[\text{with_att_syn_rules}\]
\[\text{[("relnouncla_val","is",reorder1,[("snouncla_val", fst_part),}\]
\[\text{("relpron_val", snd_part),}\]
\[\text{("joinvbph_val", thrd_part))]}\]
\[\text{with_no_inh_att_calc_rules}\]
\[\text{with_no_failure_rules}\]

$\text{orelse}\$
\[\text{consists_of [snouncla]}\]
\[\text{with_att_syn_rules}\]
\[\text{[("relnouncla_val","is",conv3,[("snouncla_val", fst_part)])]}\]
\[\text{with_no_inh_att_calc_rules}\]
\[\text{with_no_failure_rules}\]

\[\text{nouncla = consists_of [relnouncla, nounjoin, nouncla]}\]
\[\text{with_att_syn_rules}\]
\[\text{[("nouncla_val","is",reorder2,[("relnouncla_val", fst_part),}\]
\[\text{("nounjoin_val", snd_part),}\]
\[\text{("nouncla_val", thrd_part))]}\]
\[\text{with_no_inh_att_calc_rules}\]
\[\text{with_no_failure_rules}\]

$\text{orelse}\$
\[\text{consists_of [relnouncla, relpron, linkingvb, nouncla]}\]
\[\text{with_att_syn_rules}\]
\[\text{[("nouncla_val","is",reorder3,[("relnouncla_val", fst_part),}\]
\[\text{("relpron_val", snd_part),}\]
\[\text{("nouncla_val", frth_part))]}\]
\[\text{with_no_inh_att_calc_rules}\]
\[\text{with_no_failure_rules}\]

$\text{orelse}\$
\[\text{consists_of [relnouncla]}\]
\[\text{with_att_syn_rules}\]
\[\text{[("nouncla_val","us",conv4,[("relnouncla_val", fst_part)])]}\]
\[\text{with_no_inh_att_calc_rules}\]
\[\text{with_no_failure_rules}\]
The interpreter for transitive verb phrases returns two attributes for each phrase recognized. The first of these is a "transvbph_val" attribute, which is the entityset denoted by the transitive verb phrase. The second attribute returned is a set of possible subjects that could be used to resolve pronouns later in the query. These possible subjects are obtained from the possible subjects of the "jointermph" part of a transitive verb phrase.
with_att_syn_rules
[('transvbph_val','is',drop3rd,[("linkingvb_val",fst_part),
  ("passtrvb_val",snd_part),
  ("prep_val",thrd_part),
  ("jointermph_val",frth_part))],
('poss_subjs','is',same,[('poss_subjs',frth_part)])]

with_no_inh_att_calc_rules
with_no_failure_rules

The interpreter for verb phrases also returns two attributes.

The first is of type "verbph_val" and the second is of type
"poss_subjs". In DMSG, we only compute possible subjects
from proper nouns. Hence, the set of possible subjects for
verb phrases comprising of either an “intransvb” or a
"linkingvb, det, nouncla” is empty. This empty set is
obtained by applying the function “make_subjs” to the
empty list [].

verbph = consists_of [transvbph]
with_att_syn_rules
[('verbph_val','is',conv6,[('transvbph_val',fst_part)]),
('poss_subjs','is',same,[('poss_subjs',fst_part)])]
with_no_inh_att_calc_rules
with_no_failure_rules
$orelse
consists_of [intransvb]
with_att_syn_rules
[('verbph_val','is',conv13,[('intransvb_val',fst_part)]),
('poss_subjs','is',make_subjs,[])]
with_no_inh_att_calc_rules
with_no_failure_rules
$orelse
consists_of [linkingvb,det,nouncla]
with_att_syn_rules
[('verbph_val','is',applyvbph,
  [('nouncla_val',thrd_part)])],
('poss_subjs','is',make_subjs,[])]
with_no_inh_att_calc_rules
with_no_failure_rules

termph = consists_of [pnoun]
with_att_syn_rules
[('termph_val','is',conv7,[('pnoun_val',fst_part)]),
('poss_subjs','is',make_subjs,[('pnoun_val',fst_part),
  ('sort',fst_part)])]
with_no_inh_att_calc_rules
with_no_failure_rules
The specification of the interpreter for joined term phrases is the first to make use of inherited attributes. If the joined term phrase consists of a pronoun, the set of possible subjects that is in the context when the "jointermph" interpreter is invoked, is passed through as an inherited attribute to the "resolve_pronoun" interpreter. We see later how pronoun are resolved.

\[
\text{jointermph} = \begin{cases} 
\text{consists_of [termph]} \\
\text{with_att_syn_rules} \\
[\{ \text{"jointermph_val"}, \text{"is"}, conv9, \{ \text{"termph_val"}, \text{fst}_\text{part} \} \}, \\
\{ \text{"poss}_\text{subjs"}, \text{"is"}, same, \{ \text{"poss}_\text{subjs"}, \text{fst}_\text{part} \} \}] \\
\text{with_no_inh_att_calc_rules} \\
\text{with_no_failure_rules} \\
\text{orelse} \\
\text{consists_of [termph,termphjoin,jointermph]} \\
\text{with_att_syn_rules} \\
[\{ \text{"jointermph_val"}, \text{"is"}, appjoin1, \{ \text{"termph_val"}, \text{fst}_\text{part} \}, \\
\{ \text{"termphjoin_val"}, \text{snd}_\text{part} \}, \\
\{ \text{"jointermph_val"}, \text{thrd}_\text{part} \} \}, \\
\{ \text{"poss}_\text{subjs"}, \text{"is"}, app\_subjects, \{ \text{"poss}_\text{subjs"}, \text{fst}_\text{part} \}, \\
\{ \text{"poss}_\text{subjs"}, \text{thrd}_\text{part} \} \}] \\
\text{with_no_inh_att_calc_rules} \\
\text{with_no_failure_rules} \\
\text{orelse} \\
\text{consists_of [resolve\_pronoun]} \\
\text{with_att_syn_rules} \\
[\{ \text{"jointermph_val"}, \text{"is"}, conv14, \{ \text{"pronoun_val"}, \text{fst}_\text{part} \} \}] \\
(\text{with_inh_att_calc_rules} \\
[\{ \text{"poss}_\text{subjs"}, \text{"is"}, same, \{ \text{"poss}_\text{subjs"}, lhs \} \}] \\
\text{with_no_failure_rules} \\
\end{cases}
\]

\[
\text{joinvbph} = \begin{cases} 
\text{consists_of [verbph]} \\
\text{with_att_syn_rules} \\
[\{ \text{"joinvbph_val"}, \text{"is"}, conv0, \{ \text{"verbph_val"}, \text{fst}_\text{part} \} \}, \\
\{ \text{"poss}_\text{subjs"}, \text{"is"}, same, \{ \text{"poss}_\text{subjs"}, \text{fst}_\text{part} \} \}] \\
\text{with_no_inh_att_calc_rules} \\
\text{with_no_failure_rules} \\
\text{orelse} \\
\end{cases}
\]
The interpreter "sent" returns two attributes for each successful parse. The first is of type "sent_val" and the second is of type "poss_subjects". For example:

```
sent [] (words "Hall discovered phobos"). ⇒
[([SENT_VAL True,
   POSS_SUBJS [[PNOUN_VAL (function),SORT "anim",NUMBER "singlr"],
   [PNOUN_VAL (function),SORT "inanim",NUMBER "singlr"]]],
  ["."])]
```

There are two possible subjects that could be used to resolve a pronoun in a subsequent sentence. The first is associated with "Hall" the second with "phobos". Each subject is represented by a list of three attributes: a value, a sort, and a number. The sort and number help to resolve the pronoun and determine which value should replace it.

```
sent = consists_of [jointermph.joinvbph]
with_att_syn_rules
[(["sent_val","is","resolve",(["jointermph_val",fst_part),
                ("joinvbph_val",snd_part)]),
  (["poss_subjs","is","app_subjects",(["poss_subjs",fst_part),
                ("poss_subjs",snd_part)])]
with_inh_att_calc_rules
[(["poss_subjs","is","same",(["poss_subjs",lhs])]), []]
with_no_failure_rules
```

The interpreter "two_sent" passes the possible subjects from the first sentence as context to the second sentence. If the second sentence contains a pronoun then the possible subjects are used to resolve that pronoun. For example:

```
two_sent [] (words "Hall discovered phobos and he spins"). ⇒
[([SENT_VAL False], ["."])]
two_sent [] (words "Hall discovered phobos and it spins"). ⇒
[([SENT_VAL True], ["."])]
```
Pronouns are resolved in DMSG in a simplistic way. The first possible subject that has the appropriate sort and number is used to replace the pronoun. Clearly this is too simplistic to be of value other than as an example of how inherited attributes can be used in the attribute grammars that our system supports.

DMSG allows six different types of question. In each case the result returned is a set of strings, one for each successful parse of the question.
with_no_inh_att_calc_rules
with_no_failure_rules
$orelse
consists_of [quest2,joinvbph,terminator]
with_att_syn_rules
[ ("quest_val","is",ans2, [("quest2_val",fst_part),
                           ("joinvbph_val",snd_part),
                           ("sort",fst_part)]) ]
with_no_inh_att_calc_rules
with_no_failure_rules
$orelse
consists_of [quest3,nouncla,joinvbph,terminator]
with_att_syn_rules
[ ("quest_val","is",ans3, [("quest3_val",fst_part),
                           ("nouncla_val",snd_part),
                           ("joinvbph_val",thrd_part)]) ]
with_no_inh_att_calc_rules
with_no_failure_rules
$orelse
consists_of [quest4,nouncla,joinvbph,terminator]
with_att_syn_rules
[ ("quest_val","is",ans4, [("quest4_val",fst_part),
                           ("nouncla_val",snd_part),
                           ("joinvbph_val",thrd_part)]) ]
with_no_inh_att_calc_rules
with_no_failure_rules

quest4 =
consists_of [quest4a,quest4b]
with_att_syn_rules
[ ("quest4_val","is",same,[ ("quest4_val",fst_part)]) ]
with_no_inh_att_calc_rules
with_no_failure_rules

**THE SEMANTICS—PART I: The attribute evaluation and conversion functions**

**ATTRIBUTE EVALUATION FUNCTIONS**

Each attribute evaluation function is a function from a list
of attributes to a single attribute. For example, the function
"intrsctl" which is used in the "snouncla" interpreter, takes
a list containing an "adjs_val" and a "cnoun_val" attribute,
and returns a single "snouncla_val" attribute.

same [x] = x
intrsctl [ADJS_VAL x,CNOUN_VAL y] = SNOUNCLA_VAL (intersect x y)
intrsct2 \[\text{ADJ\_VAL } x, \text{ADJS\_VAL } y\] = \text{ADJS\_VAL } (\text{intersect } x \ y)
applydet \[\text{DET\_VAL } x, \text{NOUNCLA\_VAL } y\] = \text{DETPH\_VAL } (x \ y)
applytransvb \[\text{TRANSVB\_VAL } x, \text{JOINTERMPH\_VAL } y\] = \text{TRANSVBPH\_VAL } (x \ y)
applyvbph \[\text{NOUNCLA\_VAL } z\] = \text{VERBPH\_VAL } z
appjoin1 \[\text{TERMPH\_VAL } x, \text{TERMPHJOIN\_VAL } y, \text{JOINTERMPH\_VAL } z\] = \text{JOINTERMPH\_VAL } (y \ x \ z)
appjoin2 \[\text{VERBPH\_VAL } x, \text{VBPHJOIN\_VAL } y, \text{JOINVBP\_VAL } z\] = \text{JOINVBP\_VAL } (y \ x \ z)
reorder1 \[\text{SNOUNCTRA\_VAL } x, \text{RELPRON\_VAL } y, \text{JOINVBP\_VAL } z\] = \text{RELNOUNCLA\_VAL } (y \ x \ z)
reorder2 \[\text{RELNOUNCLA\_VAL } x, \text{NOUNJOIN\_VAL } y, \text{NOUNCLA\_VAL } z\] = \text{NOUNCLA\_VAL } (y \ x \ z)
reorder3 \[\text{RELNOUNCLA\_VAL } x, \text{RELPRON\_VAL } y, \text{NOUNCLA\_VAL } z\] = \text{NOUNCLA\_VAL } (y \ x \ z)
drop3rd \[\text{LINKINGVB\_VAL } w, \text{PASSTR\_VAL } x, \text{PREP\_VAL } y, \text{JOINTERMPH\_VAL } z\] = \text{TRANSVBPH\_VAL } (x \ y)
resolve \[\text{JOINTERMPH\_VAL } x, \text{JOINVBP\_VAL } y\] = \text{SENT\_VAL } (x \ y)
pick\_subject \[\text{POSS\_SUBJS } ps, \text{SORT } s, \text{NUMBER } n\] = 
\[\text{PRONOUN\_VAL } pv\]
where \(\text{PNOUN\_VAL } pv\) = \text{fst } [x \ [x,y,z] \rightarrow ps; y = \text{SORT } s; z = \text{NUMBER } n]
make\_subs \[] = \text{POSS\_SUBJS } []
make\_subs \[\text{PNOUN\_VAL } x, \text{SORT } y\] = 
\[\text{POSS\_SUBJS } [[\text{PNOUN\_VAL } x, \text{SORT } y, \text{NUMBER } "\text{single}"]\]
app\_subjects \[\text{POSS\_SUBJS } x, \text{POSS\_SUBJS } y\] = \text{POSS\_SUBJS } (x \ y)

sent\_val\_comp \[\text{SENT\_VAL } s1, \text{SENTJOIN\_VAL } f, \text{SENT\_VAL } s2\] = 
\[\text{SENT\_VAL } (f \ s1 \ s2)\]

**ATTRIBUTE "TYPE" CONVERSION FUNCTIONS**

Some of the attribute evaluation functions simply convert
the single attribute in a list to be of another type.

\[
\begin{align*}
\text{conv0} & \quad [\text{VERBPH\_VAL } x] = \text{JOINVBPH\_VAL } x \\
\text{conv1} & \quad [\text{CNOUN\_VAL } x] = \text{SNOUNCTRA\_VAL } x \\
\text{conv2} & \quad [\text{ADV\_VAL } x] = \text{ADJS\_VAL } x
\end{align*}
\]

The function "conv\_func" is used in the dictionary together
with the "meaning\_of" function. It makes the dictionary more
legible.

\[
\text{conv\_func} = [\text{"cnoun\_val"}, \text{"nouncla\_val"}, \text{conv10}, \\
\text{"intransvb\_val"}, \text{"verbph\_val"}, \text{conv11}, \\
\text{"indefpron\_val"}, \text{"detph\_val"}, \text{conv12}]\]

**FAILURE RULE FUNCTIONS**

\[
\text{check\_for\_clash } [\text{SORT } x, \text{SORT } y] = \text{False, } x = \text{"any"} \\
\quad = \text{False, } y = \text{"any"} \\
\quad = \text{False, } x = y \\
\quad = \text{True, otherwise}
\]
<table>
<thead>
<tr>
<th>QUESTION ATTRIBUTE EVALUATION FUNCTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most of the attribute evaluation functions in the “question” interpreter are used to convert results to strings that are appropriate for output to the user.</td>
</tr>
</tbody>
</table>

\[
\text{truefalse} \ [\text{SENT\_VAL} \ x] = \text{QUEST\_VAL} ("true."), \ x = \text{QUEST\_VAL} ("false."), \text{otherwise}
\]

\[
\text{yesno} \ x = \text{"yes."}, \ x = \text{"no."}, \text{otherwise}
\]

\[
\text{ans1} \ [\text{QUEST1\_VAL} \ x, \text{SENT\_VAL} \ y] = \text{QUEST\_VAL} (x, y)
\]

\[
\text{ans2} \ [\text{QUEST2\_VAL} \ x, \text{JOINVBPH\_VAL} \ y, \text{SORT} \ z] = \text{QUEST\_VAL} (x, y, z)
\]

\[
\text{ans3} \ [\text{QUEST3\_VAL} \ x, \text{NOUNCLA\_VAL} \ y, \text{JOINVBPH\_VAL} \ z] = \text{QUEST\_VAL} (x, y, z)
\]

\[
\text{ans4} \ [\text{QUEST4\_VAL} \ x, \text{NOUNCLA\_VAL} \ y, \text{JOINVBPH\_VAL} \ z] = \text{QUEST\_VAL} (x, y, z)
\]

\[
\text{whowhatq} \ x = \text{whoq} \ y, \ x = \text{"anim"}
\]

\[
\text{whos} = \text{check "nobody"}. \ [\text{name\_of} \ e | e \leftarrow \text{xs}; \text{member} (\text{set\_of\_men} + \text{set\_of\_woman}) \ e]
\]

\[
\text{whatq} \ xs = \text{check "nothing"}. \ [\text{name\_of} \ e | e \leftarrow \text{xs}]
\]

\[
\text{whichq} \ xs \ ys = \text{check "none"}. \ [\text{name\_of} \ e | e \leftarrow \text{intersect} \ xs \ ys]
\]

\[
\text{howmanyq} \ xs \ ys = \text{number} \ (\# \ (\text{intersect} \ xs \ ys))
\]

\[
\text{check str wds} = \text{str, wds = []}
\]

\[
\text{name\_of} \ e = \text{hd} \ [\text{name} | (\text{name}\_\text{[P\_NOUN\_VAL f, y]}) \leftarrow \text{p\_noun\_list}; \ f \ [\text{e}]]
\]

\[
\text{number} \ n = [\text{"none."}, \text{"one."}, \text{"two."}, \text{"three."}, \text{"four."}, \text{"five."}, \text{"six."}, \text{"seven."}, \text{"eight."}, \text{"nine."}, \text{"ten."}, \text{"eleven."}, \text{"twelve."}] \ n
\]

<table>
<thead>
<tr>
<th>THE SEMANTICS—Part II: Functions used to obtain objects denoted by proper nouns, verbs, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>FUNCTION USED TO DEFINE OBJECTS ASSOCIATED WITH PROPER NOUNS</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>test_wrt \ e \ s = \text{member} \ s \ e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FUNCTION USED TO DEFINE MEANING OF VERBS IN TERMS OF RELATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>trans_verb \ rel \ p = [x</td>
</tr>
</tbody>
</table>

| passtr\_verb \ rel = trans\_verb \ (invert \ rel) |
 FUNCTIONS DENOTED BY TERMPHRASE CONJOINERS

termph_\_and p q = g where g x = (p x) & (q x)
termph_\_or p q = g where g x = (p x)\lor (q x)

 FUNCTIONS DENOTED BY SENTENCE CONJOINERS

sand True True = True
sand any any’ = False

 FUNCTIONS DENOTED BY DETERMINERS

function_denoted_by_\_a xs ys = #( intersect xs ys ) > 0
function_denoted_by_every xs ys = includes xs ys
function_denoted_by_none xs ys = #( intersect xs ys ) = 0
function_denoted_by_one xs ys = #( intersect xs ys ) = 1
function_denoted_by_two xs ys = #( intersect xs ys ) = 2

 ************************************************************************
I I THE SEMANTICS—PART III: The “database”
************************************************************************

The database comprises a number of unary and binary relations over the set of entities that constitute the universe of discourse.

 THE UNIVERSE OF DISCOURSE

solar_entityset = [8..70]

 SETS DENOTED BY COMMON NOUNS

set_of_sun = [8]
set_of_planet = [9..17]
set_of_moon = [18..53]
set_of_men = [54..70]
set_of_women = []
set_of_thing = [8..70]

 SETS DENOTED BY ADJECTIVES

set_of_red = [12, 13, 14, 22]
set_of_blue = [11, 14, 15, 16]
set_of_depressed = [54]
set_of_green = [11, 15, 16]
set_of_brown = [9, 10, 17]
set_of_ringed = [13, 14, 15, 16]
set_of_gaseous = [13, 14, 15, 16]
set_of_solid = (union set_of_planet
set_of_moon) --- set_of_gaseous

set_of_atmospheric = [10, 11, 12, 22, 42]

set_of_vacuumous = (union set_of_planet
set_of_moon) --- set_of_atmospheric

| | SETS DENOTED BY INTRANSITIVE VERBS

set_of_exist = solar_entityset
set_of_spin = [8..53]

| | BINARY RELATIONS

rel_orbit = [(9,8),(10,8),(11,8),(12,8),(13,8),
(14,8),(15,8),(16,8),(17,8),(18,11),
  etc.

rel_discover = [(7,18),(54,19),(54,20),(55,21),(56,22),
  (56,23),(56,24),(56,25),(57,26),(57,34),
  (58,27),(58,29),(59,28),(59,30),(59,31),
  etc.

| | ****************************************

7. Concluding comments

We have presented a new method for implementing interpreters in functional languages as executable specifications. We have presented the method through an example which also demonstrates that some of Montague's ideas can be used to advantage in the construction of natural language interfaces to databases.

Clearly, our method would benefit from a syntax directed editor which would allow executable specifications of attribute grammars to be constructed through an interactive interface which guides the user, fills in parts of the syntax, and prevents errors that might otherwise be made. The method would also benefit from a more structured representation of attributes and the relationships between attributes. We are in the process of building a syntax directed editor using the Cornell Synthesizer Generator (Reps & Teitlebaum, 1989), and have begun to look at various formalisms, such as "frame structures" for the representation of attribute values.

Since this paper was submitted, we have revised the library functions such that: (i) the syntax used for specifying the attribute grammars is much 'cleaner' and is now very similar to conventional notation for attribute grammars, and (ii) fully general attribute dependencies can be specified enabling context sensitive interpreters to be built.

References


