1986 c

DSA – A Tool for Descriptive Text Analysis

A. BLAIR AND I. SOMMERVILLE

Department of Computer Science, University of Strathclyde, Glasgow G1 1XH

This paper describes a tool for descriptive text processing which can, after tuition by the user, split entity descriptions into their constituent parts and associate a meaning with each of these parts. The program, in essence, is a form of parser generator where the class of texts which may be parsed are those descriptions which are set out in a semi-structured way. Examples of such descriptions are catalogue entries in an electronic components catalogue, descriptions of flora in a handbook of wild flowers and descriptions of drugs in a formulary.

Our system is novel in that the user need not analyse the structure of descriptions in advance and then describe that structure in a formal way. Rather, DSA (Description Structure Analyser) dynamically 'learns' the structure of descriptions by interacting with the user, who tells DSA what each part of a typical description means. The description model built by DSA is then refined by seeking user advice whenever an unfamiliar description occurs. The system is not restricted to the processing of rigidly structured descriptions. It can create a parser to analyse descriptions which have a variable structure and can even tackle simple sentences of natural language.

Received July 1984

1. INTRODUCTION

An immense amount of valuable information is available in the form of semi-structured entity descriptions. Examples of entity descriptions include descriptions of the characteristics of electronic components, descriptions of drugs, descriptions of natural flora and fauna and descriptions of the facilities available to the user of a software system. In general, the user of these descriptions is presented with a book containing many descriptions and searches this manually to find the entities which are of interest.

Clearly, it would be very valuable to have this descriptive information accessible via some automated information retrieval system. Systems, such as LEXIS, a legal information system, are available where the text of the description is held on a computer as well as on paper. The user queries the description database using a keyword-based information retrieval mechanism.

Unfortunately, keyword-based systems suffer from major deficiencies because they have no knowledge of the semantics of the text, as follows.

(1) The user must be aware of the synonyms which might be used in the system for a particular keyword.

(2) The information retrieval system cannot handle cross-references amongst descriptions. If entity A is described as 'like B', a search made using B's characteristics does not normally retrieve A.

(3) The information retrieval system cannot take keyword contexts into account. A search made using a keyword 'bramble' returns descriptions containing the phrase 'unlike the bramble'.

The application of natural language processing techniques to descriptive text makes it possible to make some assessment of the meaning of that text and, in principle, improve upon the performance of keyword-based systems. Sager[1] has built a system which processes specific kinds of medical record and builds a patient database system. Other work in text processing has also been reported by Reeker *et al.*[2] and Granger *et al.*[3]

Our aim is to build an intelligent information retrieval and description processing system which can process entity descriptions, build a description database and provide the user with intelligent database interrogation facilities. This system will have four major components, as follows.

(1) A **description structure analyser.** This program makes use of the fact that descriptions are usually organised in a structured way and that some information is provided implicitly by the structuring of the description.

(2) A **semantic analyser.** This program analyses the actual text of each description and abstracts the meaning from it.

(3) A **database generator.** This program takes the output from the semantic analyser and generates a database from it.

(4) **An intelligent interrogation system.** This program is an information retrieval system which abstracts the meaning of user queries and which generates commands to retrieve entity descriptions from the database which satisfy the user's request.

An early version of the semantic analyser has been built,[4] as has a database generator which outputs Prolog clauses. Work on an intelligent retrieval system for software components is in progress. This paper is concerned with the description structure analyser (DSA).

The basis of this tool is that descriptions conform to a standard format. This need not be rigid but, in the diversity of possible description organisations there must be some fundamental structure. This structure is important as far as understanding the description is concerned as it incorporates implicit information about the entity being described. Important structural features include description layout, punctuation, typography and the use of keywords in particular positions.

For example, consider the following partial descriptions taken from a handbook describing common British mushrooms:

TUBER AESTIVUM Vitt. Truffle
CANTHARELLUS INFUNDIBULIFORMIS Fr.
LEPISTA SAEVA (TRICHOLOMA) Orton. Blewit

The scientific name of the mushroom is always given in upper-case letters and this is followed (not necessarily directly) by the naming authority. The naming authority is identifiable by the fact that it is always terminated with a full stop and appears on the same line as the scientific name. Between the scientific name and the naming authority may appear a bracketed name which is also in upper-case letters. Fungus taxonomy is not an exact science, and this is an alternative name where the particular species of the mushroom is disputed. If a common name for the mushroom exists, it follows the naming authority on the same line.

The objective of our work was to invent a tool which could handle any form of structured descriptions and which could process a large number of descriptions to abstract the meaning inherent in the description structure. Essentially, what was required was a form of parser generator which, given a description of a class of descriptions, would create a structure analyser for these descriptions.

Our initial approach to description processing was to adopt a comparable approach to that used in parser generation for programming languages.[5, 6] We defined a meta-language for describing descriptions and experimented with this using a number of different types of description. However, this approach lacked flexibility. As the form of descriptions was not under our control, we had to try to include every possible variation of description in the formal description description. This was impossible in all practical cases.

Our second attempt at the problem was more successful. A typical description is input and the user interacts with DSA to identify its constituent parts and to associate a meaning with them. He or she assigns a semantic name to each useful part of the description and defines the relationship between an object and its properties.

DSA builds a description 'profile' in the course of its interaction with the user. Further descriptions are input and, if they match the profile, are analysed according to that profile. If they do not match, DSA returns to user interaction mode for advice. Given more information from the user, the profile is dynamically modified to reflect the new possibility, and processing restarts. As descriptions are processed, a more complete picture of a description structure is built up by DSA.

The advantages of this interactive tuition approach are twofold.

(1) The user need not learn any formal notation for describing description structure. He or she need only know what the different parts of a description mean.

(2) The system is uniquely flexible. Any new variation in description format may be incorporated even after hundreds of descriptions have been processed. This does not affect, in any way, description processing which has already taken place.

In the remainder of this paper we describe, in outline, the fundamental principles underlying DSA and describe the implementation of the structure analyser. The current implementation of the system is in Prolog[7] and runs on ICL 2900 and DEC VAX computers.

2. SYSTEM FUNDAMENTALS

DSA attempts to build a description profile by analysing a number of descriptions and by modifying the profile dynamically to cope with new description formats as they arise. A description profile is built as a linked list of **frames**, where each frame holds information about one part of the description. For example, a profile of mushroom descriptions has a frame holding the scientific name, a frame holding the naming authority, a frame holding the common name and so on. The parts of a frame are as follows.

(1) **A type specification.** This is made up of a type class as discussed below, a type structure and a type representation.

(2) **A user assigned name.** This is a name given to the part of the description represented by that frame. This is optional – some frames such as those which hold punctuation or keywords are normally unnamed.

(3) **A link.** This component links the frame with some other part of the description. For example, if the frame holds some dimensions, the link would refer to the frame describing the form of the dimensioned object.

(4) **One or more pointers.** Because descriptions are flexible, there may be a number of alternative frame organisations. The sequential order of frames need not necessarily reflect the description structure, so pointers are used to link frame sequences.

Consider the profile which might be built to describe the initial line of a mushroom description above.

```
Frame 1
   Type: kernel; capitals
   User-id : mushroom_name
   Link : self
   Frame pointer : 2
Alternative
   Type : complex (
             kernel; capitals
                user_id : accepted_name
                link : mushroom_name
             kernel; constant "("
             kernel; capitals
                user_id : disputed_name
                link : mushroom_name
             kernel; constant ")"
             )
          User_id : mushroom_name
          Link : self
          Frame pointer : 2
Frame 2
   Type : simple, normal case
   User_id : naming_authority
   Link : key
   Frame pointer : 3, 4
Frame 3
   Type : kernel, normal case
   User_id : common_name
   Link : key
   Frame pointer : 4
Frame 4
   Type : kernel, hard LF
   User_id :
   Link : nil
   Frame pointer : 5
Frame 5...
```

The first frame holds the scientific name, which is in upper-case type and is linked to itself, which indicates that this is the description key. There is an alternative form of this frame where the alternative type structure is more complex. Notice that the components of complex types may have their own names and links.

The second frame is linked to the key and is the naming authority, which is in normal type. The pointers here refer to either the third or the fourth frame, as there need not be a common name for the mushroom. The third frame holds the common name, again linked to the key, and the fourth frame indicates that the first line of the description is always terminated with a linefeed.

2.1. Type specification

At the most primitive level, descriptions are simply strings of characters (letters, numbers, spaces, punctuation) but these are generally too elementary to be useful. Rather, we base our description processing on the following six 'kernel' types.

(1) Words – sequences of letters, numbers and selected punctuation characters such as hyphen (-) and apostrophe (').

(2) Phrases – these are unbroken sequences of two or more words which are all in the same typeface. Examples of phrases are 'LILY OF THE VALLEY', '**Wood blewit**' and 'Common starling'.

(3) Punctuation symbols – these are single or multiple character symbols such as semi-colon, colon, full stop, etc.

(4) Integers – unbroken sequences of decimal digits.

(5) Indentation – this is sometimes crucial in structuring a description so is included as a kernel type.

(6) Linefeed – like indentation, new lines are sometimes critical structuring features which have an implicit meaning.

There are many description components which are more complex than kernel types. We thus provide a type-definition mechanism which allows the user to define types by naming a sequence of kernel-type objects.

Simple types are sequences of kernel constructs which are considered as a single, indivisible entity. For example, the size of an entity might be expressed as '26 cm.'. The user might thus define a simple type **size** to be a combination of the kernel types ⟨integer⟩, ⟨word⟩ and ⟨full stop⟩. Once this has been defined and the context established, the system then automatically associates these kernel objects under the name **size**. It is not possible to access the internal components of a simple type.

Sometimes we want to consider a sequence of associated kernel or simple-type objects as a single type yet access components or fields of that type. For example, a date, such as '6th June, 1984' given as part of a description is clearly a single component but is made up of a sequence of simple types. We may wish to treat this as a single entity or to access the year component, the month component, etc. Therefore we allow **complex types** to be defined where a name is associated with a grouping of kernel types and the individual fields may be accessed.

Both simple and complex types are identified interactively by the system user. DSA splits the text into kernel-type objects and presents these to the user. He or she must then specify how these kernel types are associated to form simple or complex types.

As well as kernel, simple- and complex-type objects, a description often contains passages of unstructured narrative text. These are not intended for processing by DSA so, at any time, the user may suspend analysis. The description text is passed through DSA and tagged as unprocessed text for later semantic processing.

As an example of how the elements of a description are typed consider part of the description of an edible mushroom, taken from Jordan.[8]

BOLETUS TESTACEOSCABER Secr.

Habitat: typically under birch but also occasionally in mixed woods; prefers open woodland or fringes of woods; solitary: on soil;
Dimensions: cap 5–20cm. dia.

Part of the representation of this description fragment which shows the association of text with a frame, type structure, and user-defined type names is as follows:

KERNEL ⟨ phrase [word, word] ⟩ 'BOLETUS TESTACEOSCABER' ⟨caps⟩, MUSHROOM_NAME
SIMPLE ⟨abbreviation [word, dot] ⟩ 'Secr.' ⟨normal⟩, NAMING_BODY
KERNEL ⟨linefeed⟩ 'LF' ⟨hard⟩
SIMPLE ⟨ heading [word, colon] ⟩ 'Habitat:' ⟨bold constant⟩
NOPROCESS ⟨typically under birch...⟩ HABITAT
KERNEL ⟨linefeed⟩ 'LF' ⟨hard⟩
SIMPLE ⟨ heading [word, colon] ⟩ 'Dimensions:' ⟨bold constant⟩
COMPLEX (KERNEL ⟨word⟩ 'cap' ⟨normal constant⟩;
SIMPLE ⟨range [integer, dash, integer] '5–20' ⟨normal⟩ CAP_SIZE
SIMPLE ⟨abbreviation [word, dot] 'cm.' ⟨normal constant⟩
SIMPLE ⟨abbreviation [word, dot] 'dia.' ⟨normal constant⟩)
CAP_DIAMETER

The type is identified (KERNEL, SIMPLE, NOPROCESS or COMPLEX) and this is followed by a specification of its structure. For example, the initial kernel type is a phrase which, in this case, is made up of two words. Following the structure specification is the associated text which may be followed by a typeface specification where ⟨caps⟩ means the phrase is all in upper case, ⟨bold⟩ means that the element is in bold type, etc. This typeface specification may optionally be followed by 'constant' which specifies that the element never varies. For example, the description of the fungus's habitat is always introduced by '**Habitat:**'

The final element (upper-case in the representation above) is the name given to the element type. Thus, the first phrase is known to the user as MUSHROOM_NAME, the narrative text following the emboldened keyword **Habitat** is known as HABITAT, etc. Notice that some parts of the description, such as punctuation, have a structuring function only and do not provide information about the entity being described – these are not named.

2.2. Frame links

Frame links are included in a description profile so that relationships between objects and their properties may be recorded. This is best illustrated by a simple example:

FORD ESCORT
Colour: Blue. Upholstery: Plush, Colour Grey

Here it is clear (to humans) that the first colour refers to the car paintwork and the second to the car's upholstery. In the description processing, the frame representing the paintwork colour is linked to the key field (the car type) whereas the upholstery colour is linked to the Upholstery frame.

The frames which are set up (excluding punctuation frames) and their associated links are shown below:

Frame 1
kernel; 'FORD ESCORT'; caps: name = car_type: link = self

Frame 2
simple (constant word 'Colour', colon): name = ' ': link = nil

Frame 3
kernel; 'Blue'; normal: name = paint_colour: link = key

Frame 4
simple (constant word 'Upholstery', colon): name = ' ': link = nil

Frame 5
kernel; 'Plush'; normal: name = upholstery: link = key

Frame 6
simple (constant word 'Colour', colon): name = ' ': link = nil

Frame 7
kernel; 'Grey'; normal: name = interior_colour: link = upholstery

The provision of frame links which are set up as part of the initial profile development means that the system need not keep a complex semantic model of a car, upholstery, etc. in order to deduce the colour of the paintwork and the interior.

There are four permitted types of link which may be specified by the user.

(1) Self – if a frame is linked to itself this indicates that this is the description key.

(2) Key – the frame is linked to the frame which is the description key.

(3) Nil – the frame is not linked to anything

(4) User_id – the frame is linked to the frame whose user identifier is specified.

Notice that the notion of a unique key which identifies an entity is essential to our scheme. In all descriptions which we have examined, such keys are used.

2.3. Frame pointers

Normally, the frames in a profile are held in a linked list, and frame matching takes place by chaining down that list and matching adjacent frames. To handle alternative possibilities, frames in a profile may have one or more pointers which mark alternative frames and which allow alternative frames to be skipped. Thus if a match fails, pointer fields are examined and, if alternatives exist, these are then matched. Once a match succeeds, pointer fields are again used to select the next frame for matching. Pointers are illustrated in the general frame example above.

3. SYSTEM IMPLEMENTATION

The current implementation of DSA is a stand-alone program which processes descriptions and builds a database of descriptions. This description database may then be interrogated using a Prolog program.

The four main modules which make up the system follow.

(1) **The profile creation module.** This module builds up an initial picture of a description profile by user interaction.

(2) **The profile refinement module.** This is normally activated after an initial profile is set up. The profile is

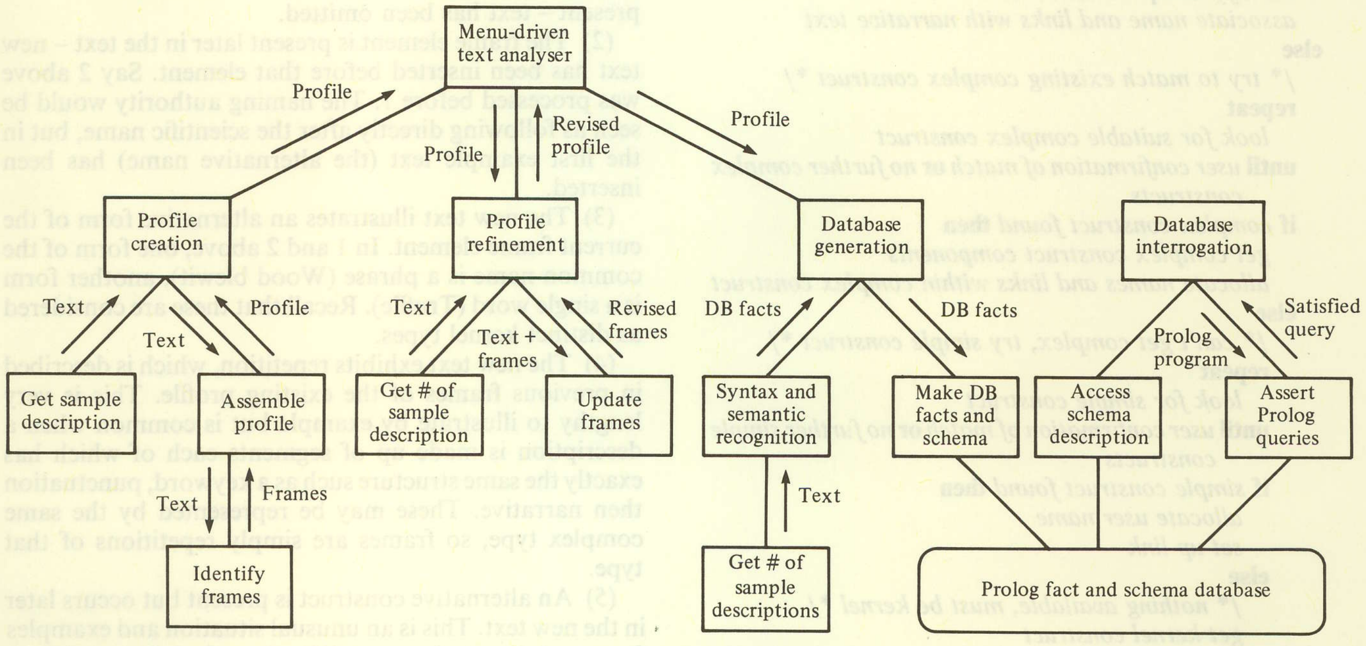

Figure 1. DSA system structure.

refined by inputting a number of other typical descriptions so that a more complete profile description may be built before information abstraction.

(3) **Database generation module.** This module processes descriptions and outputs a set of Prolog clauses which are derived from the descriptions and the profile information.

(4) **Database interrogation.** This module makes the schema of the Prolog database available to allow user-written Prolog programs to interrogate the database.

These components and their relationships are illustrated in the block diagram shown in Fig. 1.

3.1. Profile creation

The user inputs a typical entity description and this module steps through that description, picking out kernel constructs and establishing, by interacting with the user, the structure and the associated meaning of the different parts of the description.

DSA identifies kernel constructs and presents them to the user in turn. He or she is asked to confirm or deny complete syntactic representation. When a complete syntactic unit (a simple or a complex type) is identified, further dialogue establishes the status, semantic and relationship attributes of the component. When this process has been completed, the system registers this new pattern and, in so doing, starts the learning process. As further text is met, the system tries to use its accumulating knowledge to match simple and complex components which it now knows.

The system is coded in Prolog, which has inbuilt backtracking facilities. These are essential, as match failure means we must go back and try something else. This backtracking is not shown explicitly in the profile creation algorithm below.

PROFILE CREATION

```
while not end of description do
  if processing suspended then
    identify non-processable text
    associate name and links with narrative text
  else
    /* try to match existing complex construct */
    repeat
      look for suitable complex construct
    until user confirmation of match or no further complex
        constructs
    if complex construct found then
      get complex construct components
      allocate names and links within complex construct
    else
      /* can't get complex, try simple construct */
      repeat
        look for simple construct
      until user confirmation of match or no further simple
          constructs
      if simple construct found then
        allocate user name
        set up link
      else
        /* nothing available, must be kernel */
        get kernel construct
        if complete syntactic construct on its own then
          allocate name
          allocate links
        else
          /* must be a component of a larger
             simple or complex construct to be defined */
          repeat
            get kernel construct
            put kernel constructs together
          until simple or complex construct formed
          allocate name and links to construct
        end if
      end if
    end if
  end if
  store construct in profile frame
end while
```

The system adopts a 'longest possible match' strategy in that it first tries for non-processed narrative text, then complex, then simple and, when all else has failed, kernel constructs.

3.2. Profile refinement

The strategy used for profile refinement involves matching the created profile with an entity description. Each frame in the profile is matched and, if successful, the process continues until either the entire description has been processed or a match failure occurs. Match failures are common in the early refinement phase and may have several causes. To illustrate these, consider the following variations of mushroom descriptions.

1. LEPISTA NUDA (TRICHOLOMA) Cooke. Wood blewit
2. TUBER AESTIVUM Vitt. Truffle

Five distinct possible causes of frame match failure may be identified.

(1) The current frame element is not present in the new text – text has been omitted. Say 1 above was processed then 2. In the first example, there are two scientific names with the bracketed version being a disputed alternative. When the second example is processed, this is not present – text has been omitted.

(2) The frame element is present later in the text – new text has been inserted before that element. Say 2 above was processed before 1. The naming authority would be seen as following directly after the scientific name, but in the first example text (the alternative name) has been inserted.

(3) The new text illustrates an alternative form of the current frame element. In 1 and 2 above, one form of the common name is a phrase (Wood blewit), another form is a single word (Truffle). Recall that these are considered as distinct kernel types.

(4) The new text exhibits repetition, which is described in previous frames of the existing profile. This is very lengthy to illustrate by example but is common when a description is made up of segments each of which has exactly the same structure such as a keyword, punctuation then narrative. These may be represented by the same complex type, so frames are simply repetitions of that type.

(5) An alternative construct is present but occurs later in the new text. This is an unusual situation and examples from real descriptions are difficult to find. Say an initial

profile was set up using the description 'ABC lmn' and that a possible alternative for 'lmn' (a word) was a phrase such as 'pqr stu'. The initial profile would be a constant (ABC) followed by a single word. Now say a description 'ABC 123 pqr stu' was input to refine the profile. The match fails at 123 because ABC is not followed by a word but by an integer. The alternative to 'lmn' occurs later in the description. An alternative profile consisting of a constant (ABC) followed by an integer followed by phrase or a word is therefore set up.

When a match failure of any kind occurs, the user is consulted to propose possible reasons for the failure. This profile refinement process may be described algorithmically as follows.

PROFILE REFINEMENT

```
while not (end of text and end of profile) do
  if end of text and not end of profile then
    if end of profile not reachable by pointers then
      confirm text has been omitted
      insert end of profile pointer in current frame
    end if
  elsif not end of text and end of profile then
    if propose text repetition then
      insert backward pointer to repeated frame
    elsif propose inserted text then
      call profile creation to define insertion
      adjust pointers as necessary
    end if
  else /* neither end of text nor end of profile */
    attempt text match in current frame
    if no match then
      use frame pointers to try other matches
      if no match then
        if propose alternative frame construct then
            call profile creation to define alternative
        elsif propose frame match later in text then
          /* a complicated one!! */
          scan using all frames to find match
          mark match position
          back up to previous match
          /* now we know bounds of the text which won't
            match */
          create a new frame to hold text between
            matches
          adjust pointers accordingly
        elsif propose alternative frame construct later in
            profile then
          do the same as in later frame match above
        elsif propose insertion of new frame then
          call profile creation to define insert adjust
              pointers
        elsif propose possible text repetition then
          put in backward pointer to repeated frame
        else
          /*defeated – we shold never arrive here! */
          ABORT
        end if
      end if
    end if
  end if
end while
```

Profile refinement is a complex process involving a great deal of user interaction and backtracking. However, we have found that only a few descriptions need be processed during the refinement phase to establish a profile which fits almost all descriptions. Further refinement is only occasionally necessary. We have not yet encountered descriptions which cannot be represented as a profile built by the creation and refinement processes.

3.3. Database generation and interrogation

DSA is built as a stand-alone tool which generates Prolog databases. These databases are built using the refined description profile to drive a text processor which abstracts information from a description and which then outputs that information as binary tuples (Prolog clauses).

For example, consider the following description fragment:

AMANITA CITRINA Gray. False Death Cap
Habitat: woods generally; scattered solitary; on soil.
Dimensions: cap 4–10cm. dia.; . . .

The tuples generated from this fragment by DSA are as follows:

key ('AMANITA CITRINA').
mushroom_name ('AMANITA CITRINA').
naming authority ('AMANITA CITRINA', 'Gray').
common name ('AMANITA CITRINA', 'False Death Cap').
habitat('AMANITA CITRINA', 'woods generally; . . . on soil').
cap_dimensions (AMANITA CITRINA', '4–10cm. dia.').
smallest_cap_size ('4–10cm. dia.', '4').
largest_cap_size ('4–10cm. dia.', '10').
. . .

DSA also generates a schema for each type of description. This schema is presented to the user and is used when information is to be retrieved from the database. For the mushroom description, part of this schema is:

key (mushroom_name).
naming_authority (mushroom_name, _).
common_name (mushroom_name, _).
habitat (mushroom_name, _).
cap_dimension (mushroom_name, _).
smallest_cap_size (cap_dimension, _).
largest_cap_size (cap_dimension, _).
. . .

We have not yet devised any query language to access this database. The user must write his or her own Prolog program to retrieve information. For example:

Find the common name of a mushroom with cap size between 3 and 9 cm. This query might be resolved by the following procedure.

```
result (Common) : common_name (Y, Common),
                  cap_dimension (Y, Range),
                  smallest_cap_size (Range, A),
                  A > = 3,
                  largest_cap_size (Range, B),
                  B < 9.
```

First of all, this procedure finds a mushroom which has a common name and holds this in Y. It then finds the range of cap dimensions and checks that the smallest cap dimension is greater than or equal to 3. If failure occurs here, the system backs up and finds another mushroom. The same procedure is carried out for the largest cap size