INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-415-642-4274 • FAX 1-415-643-7684

L_0: A Testbed for Miniature Language Acquisition

Susan Hollbach Weber
Andreas Stolcke[1]

TR-90-010

July 1990

Abstract

L_0 constitutes a recent effort in Cognitive Science to build a natural language acquisition system for a limited visual domain. As a preparatory step towards addressing the issue of learning in this domain, we have built a set of tools for rapid prototyping and experimentation in the areas of language processing, image processing, and knowledge representation. The special focus of our work was the integration of these different components into a flexible system which would allow us to better understand the domain given by L_0 and experiment with alternative approaches to the problems it poses.

[1]Supported by an IBM Graduate Fellowship.

1 Introduction

The Miniature Language Acquisition (MLA) task is easy to describe yet difficult to solve. The task is to learn a natural language fragment given training input consisting of simple geometric scenes accompanied by partial descriptions in the language of choice. Figure 1 shows an example of an input picture, along with several partial descriptions in various languages of the picture's content. The task is to learn the grammar and semantics of the natural language, by observing the correspondences between the input pictures and their accompanying texts in that language [2].

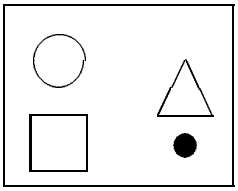
Since the project is undertaken from a computer science perspective, the end result envisioned for the project is a system that can learn the syntax and semantics of any natural language as it pertains to the given picture description task. The task is 'theory-free' and admits any implementation.

This report describes the design and capabilities of a system, written in Prolog, that performs the target task of the language acquisition problem. The inputs of the system are a natural language 'known' to the system, and a simple geometric scene. The system 'understands' the natural language description in terms of the visual input, and behaves accordingly. Although none of the system's functionality is learned, this project supplies a testbed for the eventual development of a solution to the MLA task.

1.1 L_0, the chosen MLA circumscription

Our investigation into the spatial semantics of natural languages is limited for now to the Romance and Germanic languages. We plan to eventually extend the catalogue of features as required by other language groups.

The semantic content of geometric scene description in Romance and Germanic languages is captured in the following proto-grammar known as L_0, shown in Figure 2. The L_0 description language combines semantic limits with syntactic. Two such syntactic limits are allowing only one 'far' modifier on relations, and only one level of NP conjunction. However, as language learning is not the focus of this paper, such issues will be ignored for now. It is the semantic limits imposed by L_0 that are relevant to this discussion. The relevant linguistic concepts are shape, size, shading and relative position descriptors; no attempt is made to represent object orientation. L_0 also imposes constraints on the form of the visual input, limiting scene shapes to three predefined figures and permitting only two distinct shadings.



A circle is above a square.

Un triangle est à droit d'un cercle et d'un carré.

Um cerculo escuro esta embaixo de um triangulo.

Figure 1: Training input to the L_0 language learning task: a sample picture and several possible partial descriptions in English, French and Portuguese.

S = NP | NP VP
NP = DET NP1 | DET NP1 and DET NP1
VP = VI PP | VT NP
NP1 = OBJ | SHADE OBJ | SIZE OBJ | SIZE SHADE OBJ
PP = REL NP
VI = is | are
VT = touches | touch
DET = a
OBJ = circle | square | triangle
SHADE = light | dark
SIZE = small | medium | large
REL = REL1 | far REL1
REL1 = above | below | to the left of | to the right of

Figure 2: The formal specification of the L_0 task for English.

1.2 Implementation

The MLA problem is designed to focus specifically on language *acquisition.* Hence a system that merely performs in the linguistic and visual domain defined by L_0 does not address the task as such. However, there are several good reasons for avoiding the issue of learnability for the moment and actually building a prototype system that implements only the target performance required by L_0.

For one thing, even the simple world of L_0 contains enough intricacies to merit a reassuring existence proof for what we ultimately want to achieve by a learning system. More importantly, we view our system as a testbed environment to experiment with different approaches to language processing, knowledge representation, image processing, and their interfaces. Any working learning system will be based on a feasible theory of how these areas function, and depend on the correct assumptions about underlying representations. The hope is that the Prolog system described in this report will function as a useful tool for testing various theories in cross-linguistic semantics.

These are the goals we claim our system achieves:

- Easy implementation of grammars for various languages.
- Easy implementation of a knowledge representation scheme as well its modification and extension.
- Interfacing between the above two components.
- Testing of any combination of such components on sample inputs.

We chose Prolog as the implementation language for our testbed, for the following reasons:

- Prolog's backtracking and unification mechanisms provide suitable support for the implementation of unification grammars [12].
- Interfacing between components is simplified due to the declarative nature of Prolog programming. If the nature of task is suitable, bidirectional interfaces can be implemented with

relative ease. For example, the language processing component described in section 3 does not only handle sentence analysis as required by the testbed system, but also handles generation of sentence-semantics pairs as training patterns for a connectionist approach to the L_0 task [13].

- The high-level nature of Prolog is generally supportive of rapid prototyping of various concepts for test purposes. For language and image processing in particular, Prolog's built-in control structures allowed implementations which are simple and elegant, if not optimal.

Although Prolog is not very efficient for the very low-level parts of our image-processing component (cf. section 2), this is tolerable since low-level vision is not the focus of our current efforts on L_0.

1.3 Design decisions

The MLA task as framed in [2] imposes few restrictions on the final form of the language understanding system. For example, given a picture, the system could be required to generate a complete linguistic description in the language of choice. Given a description, it could generate a picture to fit that description by arbitrarily choosing values for unspecified properties. Much simpler than either of these tasks is to give the system a picture-language pair, and ask for verification: does the sentence accurately describe the picture? This third option, which amounts to turning the description into a true/false query on the content of the picture, has been selected as the target task for our Prolog system.

A second design issue is one of internal system architecture. Given that the system should answer linguistic queries on pictorial input, what data structures and operations are needed to do this?

Some of the features we have identified as important architectural components are:

- concepts: abstract definitions of (linguistically relevant) feature clusters in the visual input. Examples of these features include shape, shade, size, position and orientation.
- objects: individual instantiations of concepts corresponding to figures in the visual scene.
- relationships between individual objects in the picture. These will include relative position (eg. left of, above, touches), relative size (eg. smaller), relative shade (eg. lighter), and relative orientation (eg. tilted, points to). Relative shape (eg. rounder, more angular) is also conceivable, although not commonly referred to in English.
- word morphology for the current language.
- grammar syntax for the current language.
- word semantics, namely, correspondence between lexical tokens and concepts for the current language.
- object reference, namely, the correspondence between lexical tokens and objects in the scene. Reference also supplies the requisite point of view for relationships between concepts. Most inter-object relationships expressed here are not commutative, so context must be supplied in the form of a reference object.

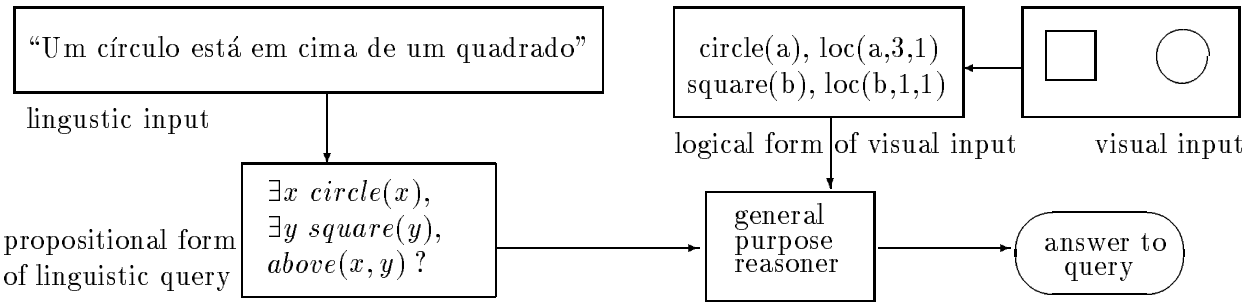


Figure 3: Coarse schematic of the principal modules in the system's architecture.

The question of object reference raises an interesting point. There is a difficulty in that visual input is inherently geometric, while language is characteristically relational. The problem is to find the relational information pointed to by the language description within the absolute information available at the level of the image. Even in the tiny domain of geometric figures, the potential number of inter-object relationships is much too large to be simply catalogued for all known natural languages. Instead, the system will need to be able to verify on demand a relationship given by a query. If relations are defined in procedural terms, the impossible task of tabulating all conceivable relationships is avoided.

The dualistic nature of the system input suggests the system can be roughly cleft in two, with visual conceptual structures one one side and linguistic structures on the other (see Figure 3). On the one hand, concepts capture all relevant visual features. Objects (concept instantiations) correspond to feature values present in the image. Machinery exists to compute on demand any relationship between two objects or groups of objects, as discussed in Section 2. On the other hand, noun semantics are grounded in objects. Verb semantics are grounded in calls to the relationship procedures. Syntax determines the order of arguments to these calls, thus supplying the requisite context. In this analysis, if the input phrase can be translated into the appropriate set of calls to the relationship mechanisms, the problem has been solved. The details of this translation process appear in Section 3.

2 Scene Processing

A graphical scene design tool, written in Prolog as an X windows application, enables the user to interactively create simple geometric scenes or scene sequences to test the system's linguistic capabilities. The tool (see Figure 2(a)) has soft buttons to create new objects (the shape menu appears in Figure 2(b)), move an existing object to a new location on the canvas, remove an object from the scene, draw or erase a coarse reference grid, "register" a scene (more on this later), clip the drawing canvas for efficiency, and pop up the description input window shown in Figure 2(c). The linguistic query thus entered is passed directly to the natural language parsers described in Section 3, and the results of the query, a 'yes' or 'no' answer, appear in a subwindow below the linguistic input.

The first meaningful action a user can perform on starting up the graphics interface is to create

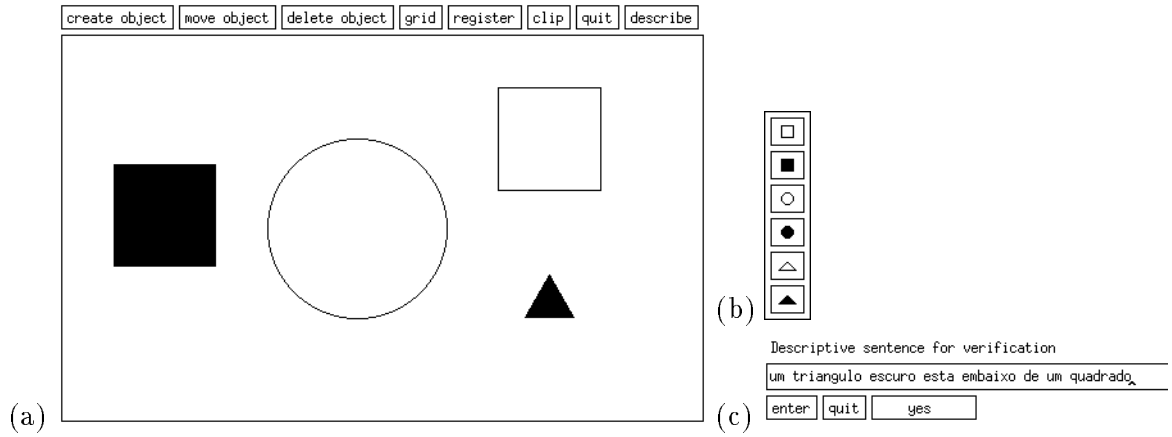


Figure 4: The graphics interface main window (a) and two pop-up windows: (b) the shape menu used to create objects and (c) the description entry window to pose queries to the system. Note that the yes/no answer to the posed question appears in the query window.

---

an object. This is done by choosing a shape from the pop-up menu, placing it on the canvas, and sizing it to the desired dimensions. A compact representation of the object corresponding to the screen image is created for future manipulation by the system. Once a scene object has been created, the user can move this object by clicking in the object and dragging it to a new location, or delete it by pointing to the object and clicking once. All menu actions can only be initiated immediately after menu selection, so clicking on empty canvas after menu selection will reset the system to a neutral state.

The user can query the system at any time, even before creating any objects (the answer to any query at this stage will, of course, be 'no'). As the grammars support isolated noun phrases, statements of fact can be verified about scenes with only one object. The interesting cases, of course, only arise in scenes of two or more objects. Before describing the kind of spatial reasoning the system can perform, however, some representational issues must first be dealt with.

2.1 Representing the picture content

One way to represent a scene is to set up data structures for both objects and relations, such as the ones given in table 1.

There are several problems with this representation. The object representation cannot directly encode size in the manner suggested, since linguistic terms such as 'large' are relative. An object of a given size will be considered large with respect to a smaller object, but small with respect to a larger object. This is a relatively minor point and can be easily fixed by altering *size* to *radius*, where the numerical value of radius is some absolute (image coordinate frame based) size metric. The truth of the assertion $large(x)$ for a given object can be tested by comparing that object's radius to other object radii. The radius field can also be used to compute the semantics of the 'far' modifier. If the distance between two objects is greater than the larger of the two object radii, then the objects are far apart.

The problems with the suggested relational representation, however, are more severe. Merely tabulating the pairwise spatial relations that hold between all objects in the picture is not ade-

object:	shape:	{circle, triangle, square}
	shade:	{light, dark}
	size:	{small, medium, large}
	origin.x, origin.y:	integer
relation:	type:	{leftof, rightof, above, below, touches}
	isfar:	boolean
	arg1:	object
	arg2:	object

Table 1: Data structures for object and relation representation (version 1).

Data structures:

object:	shape:	{circle, triangle, square}
	shade:	{light, dark}
	radius:	integer
	origin.x, origin.y:	integer

Predicates:

large, medium, small: object $\rightarrow$ boolean
larger, smaller: object $\times$ object $\rightarrow$ boolean
far: object $\times$ object $\rightarrow$ boolean
touches: object $\times$ object $\rightarrow$ boolean
above, below, leftof, rightof: object $\times$ object $\rightarrow$ boolean

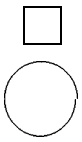
Table 2: Data structures and predicates for objects and relations (version 2).

quate to represent situations involving conjunctive relations. One form of conjunctive relation is exemplified by the phrase "a triangle is above and to the right of a circle". It is not enough to make *type* a multi-valued variable, capable of taking on both the named values at once, since this implementation is more consonant with disjunction than conjunction. Instead, the set of allowable relation types must somehow be extended to explicitly include all compound relations.

A somewhat more complex form of conjunctive relation is demonstrated in Figure 1 by the (French) sentence "a triangle is to the right of a circle and a square". This assertion is arguably not equivalent to stating that a triangle is to the right of a circle and the same triangle is also to the right of a square; rather, such a statement places the target object in relation to a newly defined region of the picture, which is the least area bounding box containing the named reference objects.

The fluidity of representation demanded by such queries led to the following re-design, in which the *relation* data structure is supplanted by one- and two-place predicates mapping objects into boolean values, summarized in table 2.

Rather than explicitly representing size in the object data structure, the object data structure



A large circle is below a square.

A small square is above a circle.

Figure 5: Changing the point of reference can change an object's size designation. In the first statement, the square is (implicitly) of medium size and the circle is large. In the second statement, however, the circle is medium sized and the square is thus considered to be small.

---

records the object *radius*. One-place predicates *large, medium* and *small* compute the sizes of objects relative to the global distribution of object radii in the image. Comparisons are made to the median object radius, where the median radius is the midpoint of a range of radii or the most frequent of two or three values (in order to capture the normative connotations of size 'medium').

This labeling scheme works well for isolated noun phrases, but does not capture the relative size comparisons available by context shifting within the picture. Depending on the reference object chosen, a given object may be considered to be of different sizes, as depicted in Figure 5. Thus we need two more predicates, *smaller* and *larger*, in which the reference object for the comparison is supplied. Queries such as the two mentioned in Figure 5 should be phrased in terms of these comparative predicates, regardless of whether the comparative is explicitly used in the natural language query.

The *far* predicate is readily computed by comparing the Euclidean distance between the two argument objects with their radii; the predicate is true for distances greater than twice the larger of the two radii.

The *touches* predicate can be implemented by constructing a boundary description for the two argument objects from their shape and size values, and comparing these for proximity.

The semantics of the four relative position predicates *above, below, rightof* and *leftof*, however, reveals a further deficiency in the representation. The predicate $above(a, b)$, for example, should supply a truth value for the proposition that object a is above object b. This measure can be obtained by computing the percentage of b's area that overlaps with the region vertically above a, and thresholding at 50%. The other three predicates are defined analogously. Thus in order to implement these predicates, we need to introduce the notion of *regions*, and define functions that will generate appropriate regions on demand, as indicated by a query.

A region is defined by its boundaries, which can be of arbitrary shape, and a point of origin. Regions are purely transitory, created on demand in the process of answering a specific query, persisting only for the duration of the query. By contrast, *objects* are permanent, persistent records of visual input features.

Although at the implementation level it seems useful to separate permanent objects from transitory regions, this distinction results in an unnecessarily baroque set of access functions. A much cleaner semantics for functions, shown in Table 3, arises from combining the representations of objects and regions into a generalized representation of objects.

An issue which has been ignored until now is the question of parameter order. Most of the predicates and functions are not commutative, so the first parameter takes on a different semantics from the second. The second parameter in a predicate such as *above* is the reference point or *landmark* for the computation, and the first parameter is the target or *trajector*. This naming

Data structures:

object:	shape:	{circle, triangle, square, region}
	shade:	{light, dark, mixed}
	radius:	integer
	origin.x, origin.y:	integer
	bounds:	<boundary description w.r.t. origin>

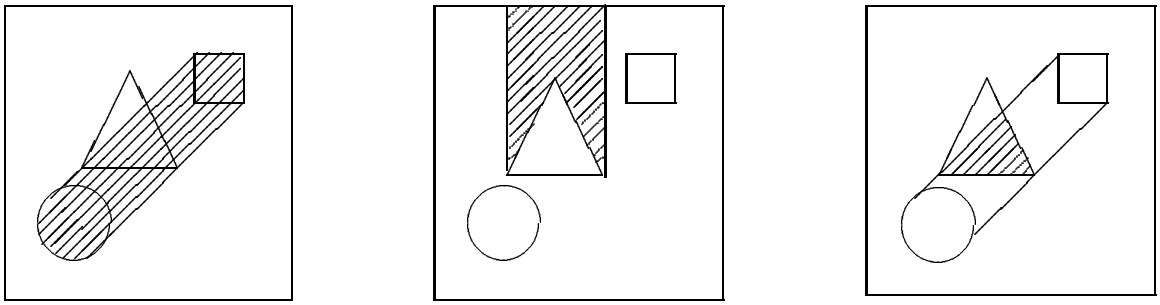
Functions:

object: variable → object instantiation
union: object × object → object
intersection: object × object → object
area: object → size metric
rAbove, rBelow, rLeftof, rRightof: object → object

Predicates:

in: object × object → boolean
above, below, leftof, rightof: object × object → boolean
large, medium, small: object → boolean
larger, smaller: object × object → boolean
far: object × object → boolean
touches: object × object → boolean

Table 3: Data structure, function and predicates for representation of objects and relations (final version).



union(square,circle) (a)

rAbove(triangle) (b)

intersection(triangle, union(square,circle)) (c)

Figure 6: Regions produced by the *union, intersection* and *rAbove* functions.

---

convention helps to keep the semantics of parameter positioning straight.

The function *union* returns a newly defined region that is the least area bounding box containing the two argument objects. This function was designed to answer conjunctive queries such as "the triangle is to the left of a circle and a square", since the query explicitly refers to the union of two objects, a circle and a square.

The function *intersection* returns the region defined by the intersection of an image object with some (previously generated) region of the image. For example, within a query one might want to manipulate the region corresponding to the union of two objects *landmark1* and *landmark2*:

$$object(trajector), object(landmark1), object(landmark2),$$
$$intersection(trajector, union(landmark1, landmark2)).$$

rLeftof (and *rRightof*) compute the area directly to the left of (and to the right of) the given object, respectively. The area is defined to be the horizontal extension of the object in the given relative direction. Similarly, *rAbove* and *rBelow* return the vertical extension of the object in the given relative direction (see Figure 6).

The functions that generate new regions are in some sense 'hidden', in that they cannot appear at the top level of a query, but only embedded within a predicate. For example, the query "the triangle is to the left of a circle and a square" could be phrased something like:

$$object(trajector), object(landmark1), object(landmark2),$$
$$trajector.shape = triangle, landmark1.shape = circle, landmark2.shape = square,$$
$$in(trajector, rLeftof(union(landmark1, landmark2))).$$

The predicate $in(trajector, landmark)$ generates a boolean confidence in the proposition that the trajector object is contained in the landmark region by computing the ratio:

$$\frac{area(intersection(trajector, landmark))}{area(trajector)}$$

This ratio is then thresholded at 50% for a boolean response. This simplistic approach is not very robust [10], but it suffices for our present purposes.

The predicate $above(trajector, landmark)$ is defined to be $in(trajector, rAbove(landmark))$. *Below, leftof* and *rightof* are defined similarly.

Area is computed from the boundary description, regardless of whether the object shape is one of the predefined primitives or not.

Most of the predicates introduced above have a natural variant in which results are in terms of a graded measure of confidence (appropriateness, fuzzy truth value) rather than a binary value. We consider our current implementation to be an approximation of such a system, which would require a fuzzified version of both the query language and the inference component, while retaining the overall structure (and the language processing component in particular). It is not clear, however, that Fuzzy Logic [14; 15] or any other proposed approximate reasoning formalism would yield the correct results in the L_0 domain.

2.2 Linguistic input processing

Thus queries to the system are phrased in terms of data structures and predicates, where the predicates are defined by functions. The system has to map the natural language descriptive texts into the logical form given by the predicates and functions. Some examples of queries and their corresponding propositional form appear below.

"a circle is below a triangle"	$\exists x, y : object(x) \wedge object(y) \wedge x.shape = circle \wedge y.shape = triangle \wedge below(x, y)$
"a circle touches a square"	$\exists x, y : object(x) \wedge object(y) \wedge x.shape = circle \wedge y.shape = square \wedge touches(x, y)$
"a circle is far to the right of a triangle"	$\exists x, y : object(x) \wedge object(y) \wedge x.shape = circle \wedge y.shape = triangle \wedge far(x, y) \wedge rightof(x, y)$
"a circle is below a small triangle"	$\exists x, y : object(x) \wedge object(y) \wedge x.shape = circle \wedge y.shape = triangle \wedge below(x, y) \wedge smaller(y, x)$
"a circle is below a dark thing"	$\exists x, y : object(x) \wedge object(y) \wedge x.shape = circle \wedge y.shade = dark \wedge below(x, y)$
"the circle below a triangle is to the left of a square"	$\exists x, y, z : object(x) \wedge object(y) \wedge object(z) \wedge x.shape = circle \wedge y.shape = triangle \wedge z.shape = square \wedge below(x, y) \wedge leftof(x, z)$

Isolated noun phrases form part of L_0. Since there is no implicit comparison with a reference object to permit relativizing the size query, the absolute form of the predicate is used.

"a small dark circle"	$\exists x : object(x) \wedge x.shape = circle \wedge x.shade = dark \wedge small(x)$

These next two queries are unable to exploit the simple relative position predicates, but are forced to use the functions directly.

"a circle and a square are below a triangle"	$\exists x, y, z : object(x) \wedge object(y) \wedge object(z) \wedge x.shape = circle \wedge y.shape = triangle \wedge z.shape = square \wedge in(union(x, y), rBelow(z))$
"a circle is below and to the left of a triangle"	$\exists x, y : object(x) \wedge object(y) \wedge x.shape = circle \wedge y.shape = triangle \wedge in(x, rLeftof(rBelow(y)))$

One class of queries which are not currently answerable is exemplified by the query "The square hides the triangle from the circle". This query presents a problem to the system since there as yet is no way to compute point-of-view trajectories. There are many other such problematic queries; the second stage of this project will be to catalogue all such queries and attempt to extend the representation proposed here to encompass them.

2.3 Implementation

The query language described in the preceding section has been implemented in Prolog. Apart from establishing minor syntactic variants on the query language presented above, the implementation is chiefly of interest for the semantic grounding it supplies for the object data type.

The one crucial representational issue avoided in the above discussion is the form taken by the object boundary description. Two options were considered. The first option is to perform two dimensional solid modeling operations on the objects as defined by their control points. A Constructive Solid Geometry [11] scheme with circles squares and triangles as the primitive shapes is the obvious approach to this task. Regions of arbitrary shape can be constructed from these primitives by creating operator trees, with operations such as union, intersection and subtraction at the nodes, and primitive shapes at the leaves of the tree. Functions are defined recursively in terms of the component primitive shapes and the results of computing the node operator on the given primitives. However, the complexity of computational geometry algorithms needed to implement such a scheme, even in only two dimensions, makes this the considerably less attractive option.

The option of choice is to represent objects in terms of bitmaps. Each object has associated with it a binary bitmap of globally fixed size. That is, the input image consisting of n objects is segmented and divided into n planes, each plane containing exactly one distinct object. The functions on arbitrary regions then have an obvious semantics in terms of region growing: the area to the right of a reference object is defined by all the pixels outside of and to the right of the object.

The query language consists of nine one-place predicates, nine two-place predicates, and seven functions. Predicates map objects into boolean truth values, while functions map objects into new objects.

2.3.1 Representing the visual scene

Before describing the predicates and functions of the query language, the method of specifying an input picture is presented. Scene content is described with the four two-place predicates `shape`, `shade`, `radius` and `origin`. For example, a light triangle at position (4,8) with radius 2 could be specified with the four facts:

```
shape([1,1], triangle).
shade([1,1], light).
radius([1,1], 2).
origin([1,1], (4,8)).
```

Conventional values for `shape` are `triangle`, `circle` and `square`; recognized values for `shade` are `light` and `dark`. (Since the existence of regions is never asserted in the Prolog database but rather computed on demand, there is no need for a `region` shape value or a `mixed` shade value; only the location and boundary description of regions are needed at the level on which they are manipulated.) `radius` and `origin` can take on any integer values between 1 and the maximum x and y values for the given drawing canvas (see the discussion of `xDim` and `yDim` below).

The identifier '`[1,1]`' is a system-generated designator that ties the four predicates together. Distinct objects must have distinct designators, and for unambiguous picture content each designator must participate in only one of each of the four predicates. That is, it is legal in Prolog to state:

```
shape(a, triangle).
shape(a, circle).
```

but the database so specified has an ambiguous representation of the shape of object 'a'. Ambiguous or incomplete object descriptions are never generated by the graphics interface; these only become issues if the user bypasses the graphical design tool and specifies objects directly to the spatial reasoning component.

The object identifier used by the system has two components, a unique object ID number, and the current 'frame' number or scene time stamp. Although not currently used by the system, the notion of frames in a time sequence is included in order to support potential future expansion of the L_0 task to incorporate image sequences. The frame number is incremented by clicking on the 'register' button of the graphics interface. The idea is to allow a user to design a picture with an arbitrary number of creates, moves and deletes, and when satisfied with the final form of the scene, to register it as a snapshot in an eventual time sequence. Since data structures corresponding to the scene visible at any given time are maintained, the user can query the system about the content of the scene at any point in this process.

The actual output of the graphics interface can now be made more precise.

- When an object is created, the four facts that define the new object are generated using the current object and frame identifiers, and the object identifier is incremented. This means that each newly created object has its own unique identifying number.

- The move operation, when executed on a current frame object, retracts the original origin fact and asserts the new origin fact.

  Moving an object defined in a previous frame results in the creation of four new facts representing the current instantiation of the object. These facts will differ only from the original facts in the identifier string (the frame number will have changed) and the new origin. There is no attempt made at the graphical level to maintain either visual or representational pointers to prior incarnations of an object; it is up to the user to remember the sequence of moves and registers that led to the current picture.

- Deleting an object retracts the four facts that define it to the system. No record is kept of the object's prior existence.

- Clipping the scene canvas can help speed up system response, since the boundary descriptions used in the spatial reasoning component are uniformly as large as the entire canvas.

- Registering a scene simply increments the time frame counter. Moving, deleting and registering are for future expansions of L_0.

- The quit button drops the user out of the graphics interface and into the Prolog substrate. Commands can be issued directly to the system at this point; the scene definitions remain in force. Reinvoking the graphics interface from the Prolog session reinitializes the object database, however, and all information from a previous scene design session is lost.

- Scene descriptions can be verified at any time during the scene design process. The system's answer to the query appears in a small subwindow of the description pop-up window.

In addition to defining all the objects in the input scene, the graphics interface defines the size of the drawing canvas that contains all the objects. This is done with the one-place facts `xDim` and `yDim`. For example, a drawing canvas that is 12 by 12 is specified by the facts:

```
xDim(12).
yDim(12).
```

2.3.2 The one-place predicates

The one-place predicates of the query language are:

`object(A)`: The query `object(A)` binds variable `A` to the data structure:

```
A = obj(Shape, Shade, Dimensions, Bitmap),
where   Dimensions = (Radius,Origin)
and     Origin = (X,Y).
```

The definition of the predicate `object` is

```
object(obj(Id, Shape, Shade, (Radius, Origin), Bitmap)) :-
        shape(Id, Shape),
        shade(Id, Shade),
        radius(Id, Radius),
        origin(Id, Origin).
```

The main use of `object` is to instantiate object variables bound by quantifiers and allow backtracking through the set of known objects. For example, given a database containing the statements

```
shape([1,1], triangle).
shade([1,1], light).
radius([1,1], 2).
origin([1,1], (4, 8)).
```

the query '`object(A)`' causes the variable `A` to be bound as follows

```
A = obj([1,1], triangle, light, (2, (4, 8)), _),
```

where the '`_`' character is used to designate an uninstantiated variable. Although the identifier string `[1,1]` does appear in the internal structure representing the unified object, this string is currently only used to unify the four facts that together define the object. The possibility exists of using the identifier to reason about the object's history between frames, as mentioned in the discussion of the 'register' button of the graphics interface, but this line of development has not yet been pursued.

`circle(A), square(A), triangle(A)`: These three predicates fill in the named shape to the first slot in the obj structure A. The query '`object(A), circle(A)`' would eventually unify the variable A with the first circular shaped object in the database (subsequent failure forces retrieval of any other objects matching this description).

The Prolog definitions of these functions are simple facts:

```
circle(obj(circle, _, _, _)).
triangle(obj(triangle, _, _, _)).
square(obj(square, _, _, _)).
```

`light(A), dark(A)`: These two predicates fill in the named shade to the second slot in the `obj` structure `A`. The query '`object(A), light(A)`' would ultimately unify `A` with the first light shaded object in the database. In Prolog, this is expressed by the facts:

```
light(obj(_, light, _, _)).
dark(obj(_, dark, _, _)).
```

`large(A), medium(A), small(A)`: These three predicates return a binary valued confidence in the size proposition stated about object `A`. The two predicates `maxMedium` and `minMedium` together define the range of radii judged to be of medium size. The definitions of `medium` and `maxMedium` appear below; `small, large` and `minMedium` are defined analogously.

```
medium(obj(_,_,_,(Radius,_,_),_)) :-
        maxMedium(MaxM),
        minMedium(MinM),
        Radius < MaxM,
        Radius > MinM.

maxMedium(X) :-
        var(X),
        median,
        maxMedium(X).

median :-
        findall(X, object(obj(_,_,_,(X,_,_),_)), Areas),
        findMax(Areas, Max),
        findMin( Areas, Min),
        findMedian(Areas, Median, Max, Min),
        Binmax is Median + (Max - Min) / 3,
        asserta(maxMedium(Binmax)),
        Binmin is Median - (Max - Min) / 3,
        asserta(minMedium(Binmin)).
```

2.3.3 The two-place predicates

The two-place predicates of the query language are:

`in(Trajector, Landmark)`: This predicate measures whether 50% or more of the trajector's area is contained in the region defined by the landmark. This predicate forms the basis for the definition of the four spatial comparative predicates, `above`, `below`, `leftof` and `rightof`.

```
in(Trajector, Landmark) :-
        intersection(Trajector, Landmark, Intersect),
        rArea(Trajector, TrajArea),
        rArea(Intersect, InterArea),
        InterArea / TrajArea > 0.5.
```

`above/below/leftof/rightof(Trajector, Landmark)`: The `above` predicate succeeds if the trajector is `in` the region above the landmark. Below, to the left of, and to the right of are defined similarly. The skeleton definition of the `above` predicate is:

```
above(Trajector, Landmark) :-
        rAbove(Landmark, RegionAbove),
        in(Trajector, RegionAbove).
```

The function `rAbove` operates on the bitmap representation of `Landmark` returning the region defined in Figure 6(b) in the `RegionAbove` parameter.

`larger/smaller(Trajector, Landmark)`: These predicates simply compare the area of the two objects. `larger` succeeds if the area of the trajector is strictly greater than the landmark; `smaller`, defined similarly, succeeds if strictly less.

```
larger(Trajector, Landmark) :-
        rArea(Trajector, TrajArea),
        rArea(Landmark, LandArea),
        TrajArea > LandArea.
```

`far(Trajector, Landmark)`: This predicate compares the Euclidean distance between the two object origins to their radii. The predicate is satisfied if the distance separating the two objects is greater than twice the larger of the two radii. Unlike the above seven predicates, which are defined in terms of functions operating on the bitmap representations of the objects, `far` operates only on the symbolic representation of the object, comparing the distance between object origins with their radii.

```
far(obj(_, _, (TrajRadius, (TrajX, TrajY)), _),
    obj(_, _, (LandRadius, (LandX, LandY)), _)) :-
        nonvar(TrajRadius), nonvar(TrajX), nonvar(TrajY),
        nonvar(LandRadius), nonvar(LandX), nonvar(LandY),
        sqrt((TrajX - LandX)*(TrajX - LandX) +
             (TrajY - LandY)*(TrajY - LandY)) > TrajRadius,
        sqrt((TrajX - LandX)*(TrajX - LandX) +
             (TrajY - LandY)*(TrajY - LandY)) > LandRadius.
```

`touches(Trajector, Landmark)`: This predicate generates the bitmap representations of the two objects and compares them pixel by pixel, succeeding on the first case encountered where a pixel in one image is adjacent to a pixel in the other image, without object overlap. `touches` relies on the bitmap representation to determine whether two pixels are adjacent but not overlapping but there is no function available to the user underlying this operation.

```
touches(Trajector, Landmark) :-
        hasBitmap(Trajector, TrajBitmap),
        hasBitmap(Landmark, LandBitmap),
        adjacentPixel(TrajBitmap, LandBitmap).
```

2.3.4 The functions

Operations that are conceptually functions of n arguments are implemented in Prolog as $(n+1)$-ary predicates, the last argument being the return parameter.

Primitive functions defined in the query language are:

`union(Object1, Object2, UnionRegion)`: This function returns a region whose only instantiated parameter is the bitmap corresponding to the convex hull of `Object1` and `Object2`, i.e.

```
UnionRegion = obj(_, _, _, _, Bitmap)}
```

where `Bitmap` corresponds to the convex hull of the two objects.

The definition of the function is:

```
union(Object1, Object2, obj(_, _, _, _, UnionRegion)) :-
        cntrlPoints(Object1, PtsObject1),
        cntrlPoints(Object2, PtsObject2),
        append(PtsObject1, PtsObject2, PointsList),
        convexHull(PointsList, UnsortedHull),
        radialSort(UnsortedHull, Hull),
        xDim(XDim),
        yDim(YDim),
        drawPolygon(x, Hull, UnionRegion, XDim, YDim).
```

The function `cntrlPoints` takes an object description (shape, radius, origin) and returns a set of boundary endpoints, such that if line segments were drawn between each (circularly) consecutive pair of points, the resulting shape would be the object in question. Circles are approximated by octagons. The convex hull of the set of control points from both argument objects is computed by taking each point and comparing it to the lines defined by all other point pairs remaining in the open list. A point must be always on one side or the other of the lines so defined for it to remain in the convex hull. If it falls between any two such lines, it is discarded. The hull points are then ordered for the polygon generation step by constructing an internal point (the midpoint of the first three non-collinear points in the list), then radially sorting the points in the list with respect to this internal point. Finally, `drawPolygon` generates the bitmap corresponding to the region defined by the control points in the convex hull.

`intersection(Object1, Object2, Region)` returns a region whose only instantiated parameter is the bitmap corresponding to the intersection of `Object1` and `Object2`, where the supporting function `intersectMaps` does the obvious (see Figure 6(c)).

```
intersection(Object1, Object2, obj(_, _, _, RegionBitmap)) :-
        hasBitmap( Object1, Object1Bitmap),
        hasBitmap( Object2, Object2Bitmap),
        intersectMaps(Object1Bitmap, Object2Bitmap, RegionBitmap).
```

`rAbove/rBelow/rLeftof/rRightof(Object, Region)`: These return a region (similar to `union` and `intersection`) whose bitmap corresponds to the appropriate image region (see Figure 6(b) for the current definition of 'appropriate').

```
rAbove(Object, obj(_, _, _, RegionBitmap)) :-
        hasBitmap(Object, ObjectBitmap),
        regionAbove(ObjectBitmap, RegionBitmap).
```

`rArea(Object, Area)` returns the pixel count `Area` of the bitmap associated with `Object`.

```
rArea(Object, Area) :-
        hasBitmap(Object, ObjectBitmap),
        computeArea(ObjectBitmap, Area).
```

2.3.5 Support functions: generating the bitmap

`hasBitmap(Object, Bitmap)` returns the bitmap corresponding to the given object description. If the object's bitmap is not yet instantiated, `hasBitmap` generates it from the symbolic description of the object. Thus bitmaps computations occur only on demand and their results are cached for future use.

`hasBitmap` uses the following three routines:

`genCircle(Object, Bitmap, N, M)` returns a `Bitmap` sized `N` by `M` containing the circle specified by `Object`, with the center point as the origin and radius as given.

`genSquare(Object, Bitmap, N, M)` returns a `Bitmap` sized `N` by `M` containing the square with origin as the bottom left corner and sides twice the radius in length.

`genTriangle(Object, Bitmap, N, M)` returns a `Bitmap` sized `N` by `M` containing the equilateral triangle with its origin as the bottom left vertex and sides twice the radius in length.

`genPolygon(Endpoints, Bitmap, N, M)` is called by `genTriangle`; it takes a list of control points and first draws the line segments defined by the sequential pairing of the points, then fills in the resulting wire frame. The first control point in the list is internally replicated as the last, in order to generate a closed figure.

2.3.6 Query language examples (revisited)

What follows are translations of English statements into queries to the spatial reasoning component. This translation is performed for various natural languages by the unification grammars described in Section 3.

Note that the existential quantifier is represented by the metapredicate `exist(X,P)`,[1] and that conjunction is expressed by the operator ',' in Prolog.

[1] All variables in Prolog are implicitly existentially quantified, so the metapredicate `exist` is trivially implemented and redundant. We use it nonetheless for explicitness and in analogy to the universal quantifier, which requires a non-trivial metapredicate definition.

"a circle is below a triangle"	`exist(X, (object(X), circle(X), exist(Y, (object(Y), triangle(Y), below(X,Y)))))`
"a circle touches a square"	`exist(X, (object(X), circle(X), exist(Y, (object(Y), square(Y), touches(X,Y)))))`
"a circle is far to the right of a triangle"	`exist(X, (object(X), circle(X), exist(Y, (object(Y), triangle(Y), far(X,Y), rightof(X,Y)))))`
"a circle is below a small triangle"	`exist(X, (object(X), circle(X), exist(Y, (object(Y), triangle(Y), below(X,Y), smaller(Y,X)))))`
"a circle is below a dark thing"	`exist(X, (object(X), circle(X), exist(Y, (object(Y), dark(Y), below(X,Y)))))`
"a circle below a triangle is to the left of a square"	`exist(X, (object(X), circle(X), exist(Y, (object(Y), triangle(Y), below(X,Y))), exist(Z, (object(Z), square(Z), leftof(X,Z)))))`
"a small dark circle"	`exist(X, (object(X), circle(X), dark(X), small(X)))`
"a circle and a square are below a triangle"	`exist(X, (object(X), circle(X), exist(Y, (object(Y), triangle(Y), exist(Z, (object(Z), square(Z), union(X,Y,XY), rBelow(Z,BZ), in(XY,BZ)))))))`
"a circle is below and to the left of a triangle"	`exist(X, (object(X), circle(X), exist(Y, (object(Y), triangle(Y), rBelow(Y,BY),rLeftof(BY,LBY), in(X, LBY)))))`

3 Language processing

3.1 Unification grammars

In the testbed system described here the language processing component has the task of producing query language expressions from input in (a subset of) some natural language. Currently this translation process is sentence-based, i.e. each sentence generates one query. Since eventually we want to be able to handle a larger number of different languages, for both analysis and generation, we needed a theoretically well-founded and sufficiently powerful computational formalism. We chose a variant of unification-based grammar [4; 5; 12], as our linguistic framework. So far we have written grammars for fragments equivalent to L_0 for English, French, German, Portuguese, and Arabic.[2] We illustrate the mechanisms of the formalism with a simple example from Portuguese.

[2]We are grateful to Terry Regier for contributing the Arabic L_0 grammar.

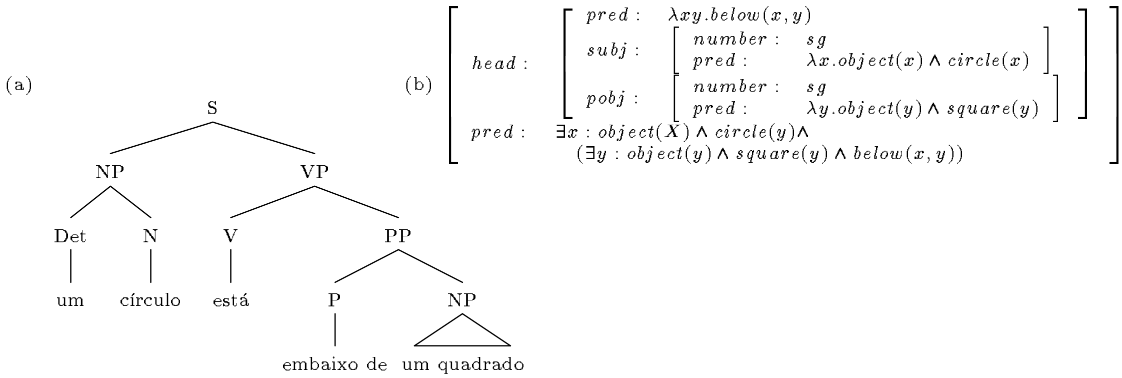


Figure 7: (a) c-structure and (b) f-structure for example sentence.

3.1.1 C-structures and f-structures

A unification-based grammar specifies languages by a two-fold structure. One component is the *constituent structure* (c-structure) of the sentence, i.e. a tree of category labels with lexical items at the leaves. C-structures are generated or parsed from standard context-free rules. As an example, consider the sentence

> um círculo está embaixo de um quadrado
> 'a circle is below a square'

which would be parsed into the c-structure depicted in Figure 7(a).

The second component of linguistic description are *functional structures* (f-structures)[3] capturing the abstract grammatical features and relations of linguistics elements (agreement, subjecthood, etc.) as well as their semantics. F-structures take the form of recursive sets of feature-value pairs, usually represented as matrices. Various features may share values, giving rise to structures that are formally equivalent to labeled directed acyclic graphs. The f-structure derived from the example sentence is shown in Figure 7(b). Note that descriptions of the sentence's subject and prepositional object have been bundled under the *subj* and *pobj* features, respectively.

3.1.2 Representing semantics

Most linguistic applications of unification grammar use f-structures primarily as a vehicle for syntactic description. For the purposes of our system, however, the main goal of the grammar is to generate a description of the semantics of a sentence. In general it is not a good idea to place the burden of semantic interpretation on a unification-based grammar, and it is usually better to devote a separate level of representation and a dedicated syntax-to-semantics mapping operator to this task. In our case we have some added flexibility due to the fact that our implementation of f-structure unification (described below) allows a natural embedding of Prolog terms in f-structures, and a corresponding coupling of the two types of unification.

This property is exploited heavily by the *pred* features in our grammar, which may hold arbitrarily complex terms. The *pred* feature is specified on all constituents carrying semantics. Iteratively

[3] The terms *c-structure* and *f-structure* were coined in the LFG-theory. We use them here to refer to their equivalents in general unification grammar formalisms, without implying that we adopt any of the additional formal machinery LFG uses.

combining *pred* features eventually yields the sentence semantics, as illustrated by the top-level *pred* value in Figure 7(b). The top-level *pred* value corresponds directly to a query-language expression.

In order to allow semantics features to be combined, *pred* values take the form of λ-*expressions*, i.e. functions of first-order terms into terms. For example, transitive verbs and prepositions typically give rise to two-place predicates. Hence their *pred* values are λ-expressions in two variables, such as this one for the preposition *embaixo* ('below', 'under'):

$$\lambda x.below(x, y).$$

Common nouns and adjectives, on the other hand, translate into unary predicates. Hence *círculo pequeno* ('small circle') translates into the *pred* feature value

$$\lambda x.object(x) \wedge circle(x) \wedge small(x).$$

(The $object(x)$ term is redundant from a declarative point of view. It is however required by the query language to enable backtracking over all objects in the database when evaluating a query.)

The purpose of using λ-expressions to describe semantics is that combining semantics can be formulated in terms of function applications of λ-expressions to other expressions. Following Montague's [6] approach we have NPs translate into λs which are applied to verb meanings (and not vice versa). This is not only necessary to handle quantification properly, but also turns to be useful in the treatment of the various semantics of noun phrase coordinations, as discussed in section 3.2. In terms of our example sentence, this means that the subject NP *um círculo* translates into a λ-expression taking a unary predicate as its argument.

$$\lambda p.\exists x : object(x) \wedge circle(x) \wedge p(x).$$

The VP *está embaixo de um quadrado* ('is below a square') translates into the complex one-place predicate

$$\lambda x.\exists y : object(y) \wedge square(y) \wedge below(x, y).$$

Combining the constituent meanings in this case involves applying the NP-λ to the VP meaning, giving (via λ-reduction) the sentence meaning

$$\exists x : object(x) \wedge circle(x) \wedge (\exists y : object(y) \wedge square(x) \wedge below(x, y))$$

3.1.3 Grammar rules

So far we have only given an account of the representational structures used in the grammar, leaving open the question how f-structures and embedded λ-expressions are constructed.

The process of f-structure construction relies on the fact that every single node in the parse tree has an associated f-structure (the sentence f-structure being the one attached to the S-node at the root). Every context-free rule has a set of constraints specifying the relations between the f-structures of the tree nodes it applies to. These constraints take the form of equations specifying which structures and substructures should be made equal, or *unified*. The top-level rule for our example would be this:

$$\begin{aligned} S \quad \rightarrow \quad & NP \quad VP \\ & S.head.subj = NP.head \\ & S.head = VP.head \\ & NP.head = p \\ & VP.head = q \\ & S.pred = p(q) \end{aligned} \tag{1}$$

By convention, grammatical features that are to be shared across various nodes of the parse tree are bundled under the feature *head*. Eventually all the syntactic information about the sentence as a whole will be available through the *head* of the S node. The first equation in the rule above simply states that that top-level *head* feature will contain the NP's *head* as the value of a subfeature *subj*. This is a formalization of the linguistic intuition that the NP figures as the grammatical subject of the sentence. The second equation then specifies that everything else in a sentence's *head* is inherited from the VP's *head*. The feature name *head* is historically motivated by X-bar theory [3], were VPs are considered the 'heads' of S's. Similar relationships (and corresponding rules) relate VP and V, NP and N, PP and P, etc.

The last three equations state the combination of λ-expressions via function application discussed in section 3.1.2. Although this operation is functional in nature it, too, can be expressed by a set of unifications, as will be shown in detail in section 3.3.3.

Lexical rules typically provide the literals making up the eventual query language expression, as in

$$\begin{aligned} P \quad \rightarrow \quad & \text{embaixo de}^4 \\ & P.pred = \lambda x.\lambda y.above(x,y) \end{aligned} \tag{2}$$

Finally we give the rule for building PPs out of Ps and NPs.

$$\begin{aligned} PP \quad \rightarrow \quad & P \quad NP \\ & PP.head = P.head \\ & PP.head.pobj = NP.head \\ & P.pred = p \\ & NP.pred = q \\ & PP.pred = \lambda x.q(\lambda y.q(x,y)) \end{aligned} \tag{3}$$

The first two equations implement *head* feature inheritance and selection of the prepositional object, similar to rule (1). The last three equations combine the two-place λ p given by the preposition and the NP-λ q into a new one-place predicate. Since the NP semantics consist of a λ that takes a one-place predicate as an argument, we apply it to the expression for P semantics $q(x,y)$ in which the first variable x is left unbound. This variable is then rebound by the new λ that gives the semantics for PP.

The rules above should give a rough idea of how the mechanisms of unification-based can be used in our testbed. More details can be obtained by consulting the references on unification grammars given above, together with the full English version of the L_0 grammar given in the appendix.

Summing up, unification equations play a double role in the formalism. Firstly, they collectively build the sentence f-structure from f-structure associated with individual constituents. Secondly, they constrain rule applications by requiring that the structures being equated be unifiable, i.e. have compatible features and sub-features. This provides an elegant account of grammatical phenomena such as agreement, coreference, control, etc. (for an overview of applications of unification in linguistic description see [12]).

The equations for *pred* we have given above show another important aspect of our formalism. There we are using both f-structure unification (denoted by the = operator) and first-order term

[4] To avoid unnecessary complexity in the syntactic domain of L_0, idioms like *to the the left of* and *embaixo de* are treated as single lexical items.

unification (implicit in the variable naming) to build the representation of the sentence semantics. Embedding term unification in f-structure unification is a non-standard extension to standard versions of unification-based grammar (which is usually based either on f-structure unification or term unification alone). The first motivation for such an extension is practical: While f-structures are naturally and conventionally used for describing syntactic entities, the same is true for first-order terms relative to logic-based semantics. A second reason for adopting this scheme is that the embedding fits naturally into our implementation of f-structure unification in Prolog, as described below in section 3.3.2.

3.2 Compositional semantics and L_0: An example

As is evident from the discussion so far, our grammar implements a version of *compositional semantics*, suffering from all the problems this approach presents. To illustrate these problems, as well as to give some further examples for the treatment of semantics in the current version of our grammars, we will discuss how *NP coordinations* are handled. The problem is that the semantics of NP coordinations is not compositional in a narrow sense, but rather depends on context. In a sentence like

a circle and a square are above a triangle

the coordination *a circle and a square* can be interpreted in at least two ways. In one case, it translates into a logical conjunction, equivalent to

a circle is above a triangle and a square is above a (the same) triangle

A second interpretation is the one where *a circle and a triangle* are construed as referring to a 'compound' region (as discussed in section 2) which then acts as the first argument to the 'above' relation. Unfortunately this second interpretation depends on the predicate involved. It is not available with, e.g., the predicate 'touch.' Thus, in the sentence

a circle and a square are touching a triangle

the NP coordination admits only the interpretation as a logical conjunction. Furthermore, note that 'touch' allows yet another interpretation for NP coordinations, namely when used as an intransitive verb:

a circle and square are touching

This kind of interpretation is restricted to verbs and prepositions expressing symmetric relations (in VPs such as *are close*, *shake hands*, etc.), and is not possible with prepositions that identify a landmark versus a trajector (*above*, *left of*, etc.).

Thus the semantics of NP coordinations depends on the semantic and syntactic context, and it is not immediately clear how these ambiguities can be handled in our framework in a non-ad hoc way. The approach currently adopted in our testbed is that we want to generate all possible interpretations for a sentence without precluding any on, e.g., pragmatic grounds. We want to ensure, however, that only interpretations which are consistent with the choice of syntax and relations involved are generated.

It seems that first-order λ-calculus semantics with the scheme of function applications suggested by Montague is reasonably well-suited to handling at least this particular problem in L_0. Under the logical 'and' interpretation for NP coordinations, a coordinate NP such as *a circle and a square* denotes a λ-expression that distributes a predicate p over a logical conjunction:

$$\lambda p.(\exists x : circle(x) \wedge p(x)) \wedge (\exists x : square(x) \wedge p(x)).$$

Expressions like this are constructed by the following NP coordination rule.

$$\begin{aligned} NP \;\rightarrow\; & NP_1 \quad NP_2 \\ & NP.head.number = pl \\ & NP.head.type = conj \\ & NP_1.pred = p \\ & NP_2.pred = q \\ & NP.pred = \lambda r.(p(r) \wedge q(r)) \end{aligned}$$

The first equation accounts for proper number agreement, while the second one marks the semantic type of the coordination.

The 'compound object' interpretation is implemented by creating a virtual object by way of the *union* function (cf. section 2).

$$\lambda p.(\exists x : circle(x) \wedge (\exists y : square(y) \wedge union(x, y, z) \wedge p(z))).$$

This is the corresponding NP coordination rule:

$$\begin{aligned} NP \;\rightarrow\; & NP_1 \quad NP_2 \\ & NP.head.number = pl \\ & NP.head.type = union \\ & NP_1.pred = p \\ & NP_2.pred = q \\ & NP.pred = \lambda r.p(\lambda x.q(\lambda y.union(x, y, z) \wedge r(z))) \end{aligned}$$

Finally, the intransitive use of verbs like *touch* with noun coordinations can be handled by simply allowing NP-λs to take two-place relations as arguments.

$$\lambda p.(\exists x : circle(x) \wedge (\exists y : square(y) \wedge p(x, y))).$$

The compound λ-expression is generated by the following NP rule.

$$\begin{aligned} NP \;\rightarrow\; & NP_1 \quad NP_2 \\ & NP.head.number = pl \\ & NP.head.type = symrel \\ & NP_1.pred = p \\ & NP_2.pred = q \\ & NP.pred = \lambda r.p(\lambda x.q(\lambda y.r(x, y))) \end{aligned}$$

One desirable feature of this approach is that the meaning of verbs like *touch* doesn't have to be changed for the intransitive interpretation (i.e. only one lexical entry is required as far as the semantics are concerned). The *pred* of *touch* continues to be a binary relation, the difference being only the way its arguments are bound in the course of rule application. The *head*-feature *type* introduced in the rules above can be used by the verbs and prepositions to properly restrict the types of conjunctions allowed as their arguments. For example, *touch* can specify that its $head.subj.type$ be from the set $\{conj, symrel\}$.

Note that conjunctive interpretations of sentences like

a circle and a square are above a triangle

also involve another type ambiguity, related to the scoping of operators and quantifiers. Its meaning can be either

$$\exists y : triangle(y) \wedge (\exists x : circle(x) \wedge above(x, y)) \wedge (\exists x : square(x) \wedge above(x, y))$$

or

$$\begin{aligned} & (\exists x : circle(x) \wedge (\exists y : triangle(y) \wedge above(x, y)) \\ \wedge \quad & (\exists x : square(x) \wedge (\exists y : triangle(y) \wedge above(x, y)) \end{aligned}$$

(The second reading is marginal in English, but seems to be somewhat more accessible in a corresponding German sentence.) This type of scope ambiguity is quite independent of the one discussed above; rather it is analogous to the one found in sentences like

all circles are above a triangle

A method for generating all possible quantifier scopes in such cases can be found in [7], and has been adapted for the L_0 testbed.

It is likely that human parsing is to some extent guided by the pragmatic context. For example, given an ambiguous construction and a visual scene it is known to refer to, some tight interactive mechanism might guide the parsing and semantic interpretation process to not even consider construals which would render the sentence false. At present our system incorporates only a very crude approximation of such an interactive mechanism. The combined parser and query interpreter will generate and try to verify every possible interpretation of a sentence in turn, until a positive outcome it reached or all possibilities are exhausted (this is a natural consequence of Prolog's backtracking capabilities). In the case of success, the satisfying parse and query are displayed, thus 'disambiguating' the sentence for the user. However, the order in which alternative parses and interpretations are considered is completely independent of the picture contents, and is thus far from being psychologically plausible.

3.3 Unification grammars in Prolog

The unification grammar formalism outlined above is clearly structured into two main components, namely a straightforward context-free (phrase structure) grammar, and the unification of f-structures for the formulation of f-structure constraints on rule application. This section explains in some detail how both these components can be realized quite elegantly in Prolog, as well as how the mapping (or 'attachment') of f-structures to c-structures is accomplished.

3.3.1 Definite Clause Grammars

Prolog incorporates a standard extension to implement phrase-structure grammars, known as Definite Clause Grammar (DCG) [8]. As a consequence the context-free part a unification grammar rule can readily be written as a DCG grammar rule in Prolog. In the case of rule (1) above this is simply

```
s --> np, vp.
```

Likewise rules (3) and (2) become

```
pp --> p, np.
p --> [embaixo,de].
```

The notational convention in DCG is that non-terminals translate into Prolog atoms, whereas terminals become list elements.

By default the Prolog system automatically compiles DCG rules into Horn clauses, which, when interpreted as standard Prolog clauses, implement a top-down, left-to-right parsing algorithm. We won't dwell on the details of this translation process; a good account can be found in [1]. It is important to note, however, that DCG is a declarative formalism which can be interpreted by several alternative parsing strategies. (See Pereira & Shieber [7] for an thorough discussion of DCG parsing algorithms.) For convenience we have built on top of the DCG implementation provided by the standard Prolog systems, although top-down, left-to-right interpretation of DCG rules via backtracking is not an optimal parsing method. The current typical sizes of our grammars allow us to do so, but a more efficient parsing technique (such as chart parsing) would be required for significantly larger systems. However, such a parser could still make use of the general f-structure unification mechanism described below.

3.3.2 F-structure unification in Prolog

Prolog supports a built-in version of unification that operates on *terms* (functor-argument structures). It would be possible in principle to rewrite an f-structure based unification grammar to use terms instead of f-structures and use term unification directly.[5] This is feasible in certain special cases, such as when the f-structures are restricted to tree-structures [9]. In our case, however, this would yield extremely unwieldy structures, at the expense of readability and the envisioned flexibility of a rapid-prototyping system.

The alternative approach is to implement f-structures as Prolog data structures and define the unification operator explicitly. This turns out to be possible in a simple elegant way which doesn't sacrifice efficiency because it still capitalizes on Prolog's built-in unification in a non-trivial way. We represent f-structures as 'open' lists of feature-value pairs, i.e. a structure

$$\begin{bmatrix} f_1: & v_1 \\ f_2: & v_2 \\ \vdots & \vdots \\ f_n: & v_n \end{bmatrix}$$

is represented as a Prolog list

```
[ f1:v1, f2:v2, ..., fn:vn | _ ]
```

with a variable `_` as its tail. (':' is defined as an infix operator so these structures can be parsed and printed in the intuitive format given above).

In the course of unifying two such lists, the tail variable gets instantiated further, thus extending the list of features as needed. Specifically, unification of two feature lists amounts to the following:

1. Add features present in only one of the lists to the other list by further instantiating the respective tail variable.

2. For features present in both lists, unify the values recursively.

[5] In fact DCG can be characterized as a special variant of general unification grammars [12].

3. After all features have been processed like this, (Prolog-)unify the two tail variables.

The last step ensures that whatever further unifications are performed on either of the two structures, any additions to the structure will be shared with all other structures that have been unified previously. This is where we crucially capitalize on term unification to prevent inefficient structure copying. The first two steps can be performed in time linear in the size of the smaller of the structures. We thus pay only a constant factor of overhead for f-structure unification compared to term unification. (As in Prolog we can do so because we dispense with the occurs check. This constitutes no limitation since the grammar theory constrains f-structures to be acyclic.)

The recursion in step 2 terminates at 'atomic' feature values. For the purpose of f-structure unification, any Prolog term that is not of the special list form introduced above is atomic. Such values are then submitted to standard Prolog unification. In most cases these values will be just Prolog atoms, in which case unification amounts to identity check. In one case, namely for *pred* feature values, we make use of the full power of term unification to implement the operations on λ-expressions required for the representation of semantics, as discussed in section 3.1.2.

It it worth pointing out that embedding term unification in f-structure unification should not be considered an ad-hoc feature of our implementation. Combining the two types of unification makes sense because both implement essentially the same abstract operation (following the same axioms). The only difference lies in the particular data structures being operated on, and can be considered a question of 'syntactic sugar.'

The f-structure unification operator outlined above has been implemented as a two-place Prolog predicate `X === Y`. It succeeds if the structures `X` and `Y` are unifiable, changing the structures accordingly (destructive unification), and fails otherwise. Unifications are undone via backtracking over `X === Y`. The next sections describe how this operator can be combined with DCG to give a full implementation of unification grammars.

3.3.3 Representing λ-expressions

Our representation of λ-expressions follows closely the approach given by Pereira & Shieber [7]. A λ-function

$$\lambda x.p(x)$$

is represented as a Prolog term

```
X^P
```

where `X` is a Prolog variable and `P` is a term containing that variable. Using Prolog variables to represent first-order variables in this context has several advantages:

- Because Prolog converts variables into unique internal identifiers the problem of variable renaming in λ-reduction is avoided.
- Identifying two formal λ-arguments can be implemented by variable unification.
- Substitution of actual arguments in λ-terms (λ-reduction) is also easily accomplished by Prolog unification: To apply the λ function `X^P` to a term `Y`, simply unify `X` and `Y` and take the body `P` (now with the properly instantiated variable `X`) as the result.

3.3.4 Putting things together

Once a general mechanism for f-structure unification is available, it becomes straightforward to add f-structures and unification equations to DCG rules. To do so, two standard features of the DCG formalism are used. Firstly, we may add a variable argument to each non-terminal to implement the attachment of f-structures to nodes in the parse tree (c-structure). Rules (1), (2) and (3) thus become

```
s(S) --> np(NP), vp(VP).
pp(P) --> p(P), np(NP).
p(P) --> [embaixo,de].
```

Here `S` will be instantiated with the f-structure attached to the `s` constituent. Prolog unifies these variables in the course of DCG rule application, thus communicating f-structures between c-structure nodes.

To actually instantiate these variables, as well as constrain rule application, we capitalize on the mechanism for 'procedural attachment' DCG provides. In DCG, arbitrary Prolog goals (procedure calls) can be added to grammar rules. These are enclosed in `{ ... }` and are executed as the rule is applied. We limit use of this feature to adding calls to the f-structure unification operator `===` introduced in the previous section. The complete DCG form of the three example rules given in section 3.1.2 is now

```
s(S) --> np(NP), vp(VP),
        { S!head!subj === NP!head,
          S!head === VP!head,
          NP!head === X^P,
          VP!head === X,
          S!head === P }.
pp(P) --> p(P), np(NP),
        { PP!head === P!head,
          PP!head!pobj === NP!head,
          PP!head!level === NP!head!level,
          P!pred === X^R,
          NP!pred === R^Q,
          PP!pred === X^Q }.
p(P) --> [embaixo,de],
        { P!pred === X^Y^below(X,Y) }.
```

Note that we use another Prolog operator, '`!`', to specify feature selection from f-structures. The operator `===` first processes feature selections on its arguments and then proceeds with unifying the resulting structures as discussed in the previous section.

The representation for λ-expressions makes the implementation of λ-reductions extremely concise, but somewhat unintuitive to read at first. For example, the three equations for *pred* features in the rule for PP accomplish the following. `P!pred` is a λ in two arguments, say `X` and `Y`, the first of which is stripped off to yield a one-argument λ `R`. `NP!pred`, a function which takes a unary predicate as an argument, is now applied to `R`, yielding a result `Q`. `Q` still contains `X`, the first argument of the preposition, as a free variable, hence we obtain the semantics of the PP by rebinding `X` in a new one-place λ-function `X^Q`.

3.3.5 Query evaluation

The top-level *pred* feature derived by a parse corresponds to a query language expression that can be evaluated directly in Prolog. The literals in the resulting expression are calls to the primitives of the query language, as described in section 2.3. Logical connectives (conjunctions, disjunctions) are represented as their Prolog counterparts and can hence be evaluated directly. Finally, quantifiers $\exists$ and $\forall$ are represented as `exist(X,P)` and `all(X,P)`, respectively, and are implemented as Prolog meta-predicates (definitions are given in the appendix).

Optionally a number of syntactic postprocessing steps can be applied to a query prior to evaluation. These include flattening of nested conjunctions and disjunctions for improved readability, elimination of constant subexpressions for efficiency, or introduction of scope ambiguities as mentioned in section 3.2.

4 Conclusion

We have described the components of a testbed system designed to assist us in our investigations into the domain of L_0, a touchstone problem defined for interdisciplinary research in cognitive science.

The current version of this system implements the target performance of the L_0 task by decomposing it into three components: a first-order logic-based query language including a set of perceptual and conceptual primitives; an implementation of this query language on the basis of a bitmap-based representation of regions; and a language processing facility performing both parsing and semantic interpretation using unification grammars.

The integration of these components was greatly facilitated by our choice of Prolog as a representation and implementation language. Its logic-based semantics provided an elegant implementation of the query language. The language processing component benefitted from a a simple and efficient embedding of both term unification in f-structure unification, and unification based grammars in the Definite Clause Grammar formalism.

References

[1] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer, 2nd edition, 1984.

[2] Jerome A. Feldman, George Lakoff, Andreas Stolcke, and Susan Hollbach Weber. Miniature language acquisition: A touchstone for cognitive science. Technical Report TR-90-009, International Computer Science Institute, March 1990. To appear in the Proceedings of the 12th Annual Conference of the Cognitive Science Society.

[3] Ray Jackendorff. *X-bar syntax.* MIT Press, Cambridge, Mass., 1977.

[4] Ronald M. Kaplan and Joan Bresnan. Lexical functional grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, Cambridge, Mass., 1982.

[5] Martin Kay. Functional unification grammar: A formalism for machine translation. In *Proceedings of the 10th International Conference on Computational Linguistics*, pages 75–78, Stanford, Calif., July 1984.

[6] Richard Montague. The proper treatment of quantification in ordinary English. In K. J. J. Hintikka, J. M. E. Moravcsik, and P. Suppes, editors, *Approaches to Natural Language. Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*, pages 221–242. Reidel, Dordrecht, Boston, 1973.

[7] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*. Number 10 in CSLI Lecture Notes Series. Center for the Study of Language and Information, Stanford, Ca., 1987.

[8] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars compared with augmented transition networks. Technical report, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, 1978.

[9] Fred Popowich. A tree unification grammar-based natural language processor. Technical Report CSS-IS TR 89-08, Centre for Systems Science, Simon Fraser University, Burnaby, B.C., Canada, December 1989.

[10] Terry Regier. *Spatial Semantics*. Dissertation proposal, Computer Science Division, University of California, Berkeley, Ca., 1990.

[11] Aristides A. G. Requicha. Representations for rigid solids: Theory, methods and systems. *ACM Computing Surveys*, 12(4), December 1980.

[12] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. Number 4 in CSLI Lecture Note Series. Center for Study of Language and Information, Stanford, Calif., 1986.

[13] Andreas Stolcke. Learning feature-based semantics with simple recurrent networks. Technical Report TR-90-015, International Computer Science Institute, Berkeley, Ca., April 1990.

[14] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, June 1966.

[15] Lotfi A. Zadeh. Fuzzy languages and their relation to human and machine intelligence. In *Proceedings of the Conference on Man and Computer, Bordeaux, France, June 1970*, pages 130–165. S. Karger, Basel, 1972.

Appendix

Quantifier meta-predicates

These meta-predicates implements the existential and universal quantifiers generated by the semantic component of the grammar.

Existential quantification is trivial since Prolog backtracks over all instantiations of free variables by default. However, we have to ensure that variables are kept uninstantiated after satisfying an existentially quantified query because variables of the same name may occur elsewhere. Hence the following definition.

```
exist( _, P ) :-
        not not P.
```

Universal quantification is always over an implication `P -> Q`. It is implemented by backtracking over all solutions for `P` and making sure that there is a matching solution for `Q`.

```
all( _, P -> Q ) :-
        P,
        ( not Q, !,
          fail
        ;
          fail
        ) ; true.
```

English L_0 grammar

The following is the full text of our current English version of the L_0 grammar. It makes extensive use of disjunctions (the ';' operator) in right-hand sides of grammar rules to factor out identical equations, non-terminals and terminals in alternative productions for a given non-terminal. We use this Prolog-specific feature to reduce the redundancies that are otherwise typical for large unification-based grammars. Also note that unification equations mostly precede non-terminals and terminals in right-hand sides of rules. This is was done to enhance efficiency in the interpretation of the grammar and has no bearing on its declarative semantics.

The syntactic coverage of this grammar goes considerably beyond the minimal L_0 grammar given in Figure 2. In particular, the following constructions are supported:

- PP as modifiers on NPs, e.g. *a circle below a triangle above a square.* Note that these constructions introduce ambiguity (*above a square* can modify either *circle* or *triangle*).
- Topicalized PPs, e.g. *below a triangle is a square.*
- General predicates (either PPs or present participle VPs) as arguments of a copula (*a circle is touching a triangle*) or modifying an NP (*a circle touching a triangle is below a square*).
- Coordinations of predicates, e.g. *a circle is below a triangle and touching a square.*
- The *all* determiner.
- Coordinations of prepositions with non-conjunctive semantics, e.g. *a circle is below and to the left of a square* (cf. section 2).

Equations relating to semantics are marked with the comment string `SEMANTICS`.

```
s(S) -->
        (
                { S!head === NP!head,
                % SEMANTICS
                  NP!pred === (X^true)^P,
                  S!pred === P },
                np(NP)
        ;
                { S!head === VP!head,
                  S!head!subj === NP!head,
                  S!head!level === NP!head!level,
                  VP!form === fin,
                % SEMANTICS
                  NP!pred === X^P,
                  VP!pred === X,
                  S!pred === P },
                (
```

```
                        { VP!inv === - },
                        np(NP), vp(VP)
                ;
                        { VP!inv === + },
                        vp(VP), np(NP)
                )
        ).

np(NP) -->
        (
                { NP!head === NP0!head,
                  NP!head!type === generic,
                  NP!pred === NP0!pred },
                np0(NP0)
        ;
                { NP!head!number === pl,
                  NP!head!conj1 === NP0!head,
                  NP!head!level === NP0!head!level,
                  NP!head!conj2 === NP1!head,
                  NP!head!level === L, inc_level(L,L1),
                  NP1!head!level === L1 },
                np0(NP0), [and], np(NP1),
                % SEMANTICS
                {
                  (        % logical and interpretation
                        NP!head!type === generic,
                        NP0!pred === R^P,
                        NP1!pred === R^Q,
                        NP!pred === R^(P,Q)
                  ;        % region union interpretation
                        NP!head!type === runion,
                        NP0!pred === (X^Q)^P,
                        NP1!pred === (Y^(rUnion(X,Y,Z),U))^Q,
                        NP!pred === (Z^U)^P
                  ;        % symmetrical relation interpretation
                        NP!head!type === symrel,
                        NP1!pred === (X^R)^P,
                        NP0!pred === (Y^P)^Q,
                        NP!pred === (X^Y^R)^Q
                  ) }
        ).


np0(NP) -->
        { NP!head === NP1!head,
          NP!head === Det!head,
        % SEMANTICS
          Det!pred === X^P,
          NP1!pred === Q,
          PRED!pred === Q^X,
          NP!pred === P },
        det(Det), np1(NP1),
        (
                { PRED!pred === Y^Y },
                []
        ;
                { NP!head!spec === PRED!head,
                  NP!head!level === L, inc_level(L,L1),
                  PRED!head!level === L1 },
```

```
                pred_list(PRED)
        ).

/* use a 'concreteness' hierarchy to account for the ordering of adjectives */
np1(NP) -->
        { NP!head === N!head },
        (
                { NP!pred === N!pred },
                n(N)
        ;
                { Adj!head === NP!head },
                adj(Adj), n(N),
                % SEMANTICS
                { N!pred === X^P,
                  Adj!pred === X^Q,
                  NP!pred === X^(P,Q) }
        ;
                { Adj1!head === NP!head,
                  Adj2!head === NP!head },
                adj(Adj1), adj(Adj2), n(N),
                { Adj1!concreteness === C1,
                  Adj2!concreteness === C2, C1 < C2,
                % SEMANTICS
                  N!pred === X^P,
                  Adj1!pred === X^Q,
                  Adj2!pred === X^R,
                  NP!pred === X^(P,Q,R) }
        ).

vp(VP) -->
        { VP!form === VP0!form,
          VP!inv === VP0!inv },
        (
                { VP!head === VP0!head,
                  VP!pred === VP0!pred },
                vp0(VP0)
        ;
                { VP!form === fin,
                  VP!form === VP1!form,
                  VP!inv === -,
                  VP!inv === VP1!inv,
                  VP!head!conj1 === VP0!head,
                  VP0!head!level === VP!head!level,
                  VP!head!conj2 === VP1!head,
                  VP!head!level === L, inc_level(L,L1),
                  VP1!head!level === L1 },
                vp0(VP0), [and], vp(VP1),
                % SEMANTICS
                { VP0!pred === X^P,
                  VP1!pred === X^Q,
                  VP!pred === X^(P,Q) }
        ).

vp0(VP) -->
        { VP!form === V!form,
          VP!head === V!head },
        (
                { V!class === cop,
                  VP!head === PRED!head,
```

```
                VP!pred === PRED!pred },
              (
                      { VP!inv === - },
                      v(V), pred(PRED)
              ;
                      { VP!inv === +,
                        PRED!top === + },
                      pred(PRED), v(V)
              )
        ;
              { V!class === intrans,
                VP!inv === -,
                VP!pred === V!pred },
              v(V)
        ;
              { V!class === trans,
                VP!inv === -,
                VP!head!obj === NP!head,
                VP!head!level === NP!head!level },
              v(V), np(NP),
              % SEMANTICS
              { V!pred === X^P,
                NP!pred === P^Q,
                VP!pred === X^Q }
        ).


pred_list(PRED) -->
        { PRED!head === PRED1!head },
        pred(PRED1),
        (
                { PRED1!pred === X^P,
                  PRED!pred === (X^Q)^(X^(Q,P)) }
        ;
                { PRED!head!spec === PRED2!head,
                  PRED!head!level === L, inc_level(L,L1),
                  PRED2!head!level === L1 },
                pred_list(PRED2),
                % SEMANTICS
                { PRED1!pred === X^P,
                  PRED2!pred === (X^(Q,P))^R,
                  PRED!pred === (X^Q)^R }
        ).

pred(PRED) -->
        { PRED!top === PRED0!top },
        (
                { PRED!head === PRED0!head,
                  PRED!pred === PRED0!pred },
                pred0(PRED0)
        ;
                { PRED!head!conj1 === PRED0!head,
                  PRED0!head!level === PRED!head!level,
                  PRED!head!conj2 === PRED1!head,
                  PRED!head!level === L, inc_level(L,L1),
                  PRED1!head!level === L1,
                % SEMANTICS
                  PRED0!pred === X^P,
                  PRED1!pred === X^Q,
```

```
                  PRED!pred === X^(P,Q) },
                pred0(PRED0), [and], pred(PRED1)
        ).

pred0(PRED) -->
        (
                { PRED!head === PP!head,
                  PRED!pred === PP!pred },
                pp(PP)
        ;
                { PRED!top === -,                % cannot topicalize
                  VP!form === ing,
                  PRED!head === VP!head,
                  PRED!pred === VP!pred },
                vp(VP)
        ).

v(V) -->
        { V!class === cop,
          V!form === fin },
        (
                { V!head!subj!number === sg },
                [is]
        ;
                { V!head!subj!number === pl },
                [are]
        ).

v(V) -->
        {
                V!class === trans,
                V!head!subj!type === generic,
                V!head!obj!type === generic
          ;
                   V!class === intrans,
                V!head!subj!type === symrel
        },
        (
                { V!form === fin },
                (
                        { V!head!subj!number === sg },
                        [touches]
                ;
                        { V!head!subj!number === pl },
                        [touch]
                )
        ;
                { V!form === ing },
                [touching]
        ),
        % SEMANTICS
        { V!pred === X^Y^touch(X,Y),
          V!head!pred ==& V!pred }.

adj(A) -->
        { A!concreteness === 1 },
        % SEMANTICS
        (
                { A!pred === X^small(X) },
```

```
                [small]
        ;
                { A!pred === X^medium(X) },
                [medium]
        ;
                { A!pred === X^large(X) },
                [large]
        ),
        { Adj!head!pred ==& Adj!pred }.

adj(A) -->
        { A!concreteness === 2 },
        % SEMANTICS
        (
                { A!pred === X^light(X) },
                [light]
        ;
                { A!pred === X^dark(X) },
                [dark]
        ),
        { Adj!head!pred ==& Adj!pred }.

n(N) -->
        { N === N1,
          N === N2 },
        n_stem(N1), n_suf(N2).

n_stem(N) -->
        % SEMANTICS
        (
                { N!pred === X^circle(X) },
                [circle]
        ;
                { N!pred === X^square(X) },
                [square]
        ;
                { N!pred === X^triangle(X) },
                [triangle]
        ),
        { N!head!pred ==& N!pred }.

n_suf(N) -->
        (
                { N!head!number === sg },
                []
        ;
                { N!head!number === pl },
                [-s]
        ).

det(Det) -->
        (
                { Det!head!number === sg,
                % SEMANTICS
                  Det!pred === (X^P)^(X^Q)^exist(X,(object(X),P,Q)) },
                [a]
        ;
                { Det!head!number === pl,
                % SEMANTICS
```

```
                  Det!pred === (X^P)^(X^Q)^all(X,(object(X),P -> Q)) },
                [all]
        ).

pp(PP) -->
        { PP!head === P!head,
          PP!head!pobj === NP!head,
          PP!head!level === NP!head!level,
        % SEMANTICS
          P!pred === X^R,
          NP!pred === R^Q,
          PP!pred === X^Q },
        p(P), np( NP ).

p(P) -->
        (
                { P!head === P0!head,
                  P!pred === P0!pred },
                p0(P0)
        ;
                % SEMANTICS
                { P1!pred === X^Y^Q,
                  P2!pred === X^Y^R,
                  P!pred === X^Y^compose(Q,R) },
                p1(P1), [and], p1(P2)
        ).

p0(P) -->
        { P!head === P1!head },
        (
                { P!pred === P1!pred },
                p1(P1)
        ;
                { M!head === P!head },
                mod(M), p1(P1),
                % SEMANTICS
                { P1!pred === Q,
                  M!pred === Q^R,
                  P!pred === R }
        ).

p1(P) -->
        % SEMANTICS
        (
                { P!pred === X^Y^above(X,Y) },
                [above]
        ;
                { P!pred === X^Y^below(X,Y) },
                [below]
        ;
                { P!pred === X^Y^leftof(X,Y) },
                [to, the, left, of]
        ;
                { P!pred === X^Y^rightof(X,Y) },
                [to, the, right, of]
        ),
        { P!head!pred ==& P!pred }.

mod(M) -->
```

```
{ M!pred === (X^Y^Q)^(X^Y^far(Q)),
  M!head!pred ==& far(Q) },
[far].
```