CASL Single Lecture Notes

SLN - Lisp - 04

2005-09-22

Cognitive Autonomous Systems Laboratory
Department of Computer and Information Science
Linköping university, Sweden

Erik Sandewall

Lisp Software Design Techniques, and the Ethel System as an Example

Subject Area: System Design in Lisp

Date of lecture: 2005-09-06, -20, and -23

"Single Lecture Notes" are notes corresponding to one or a few lectures.

Related information can be obtained via the following WWW pages:

Linköping university:	http://www.liu.se/
This course:	http://www.ida.liu.se/~TDDA23/
CASL Group:	http://www.ida.liu.se/ext/casl/
The author:	http://www.ida.liu.se/~erisa/

1 Topic

This memo is lecture notes for three lectures in the course that address the combination of two topics: some software development methods that are characteristic of Lisp, and an introduction to the concept of actions which is important in A.I. and in particular for robotic and other intelligent agent systems. The memo also contains software documentation for a demo and lab system, called Ethel, that was used in these two lectures and that is also used for lab 3 in this course.

There is not only single programming style in Lisp, no more than in any other programming language. Different people use the language in different ways. The methodology aspects of this memo therefore represent Lisp "the way I use it". My viewpoint is based on the distinguishing properties of this language compared to more conventional ones.

The following are the software techniques we want to describe and illustrate:

- The use of *self-describing constructs* in an object-oriented database (or more precisely, a concept-oriented database).
- The *multiple use of declarative information*, that is, information that specifies the structure or the contents of some body of data. In particular, we show examples of how it is used both for data management, such as for storing data persistently, and operationally in the main computational process.
- The use of *attached functions and procedures*, that is, segments of code that are kept and referenced as properties of symbols under various property-tags, instead of the usual arrangement where every function or procedure has its own, atomic name.
- The definition, implementation, and use of *embedded, special-purpose languages*, exemplified here with a simple language for situation-effect rules.

The term 'action' is used differently in different contexts. In our case we consider durative actions, that is, actions that operate during an interval of time, as well as related concepts and some implementation methods for them.

This Ethel demo and lab system, which is introduced as a concrete example, is built around a miniworld where besides people there are cities and a few other geographical types, and where airplanes take passengers and fly between airports. It has the character of a simple *base system* defining the overall structure, basic procedures such as the action simulator, and a few examples of data objects and action verbs. This is sufficient for demonstrating the concepts and design techniques mentioned above, in particular in the lectures. It is also intended as a framework or 'skeleton' where more objects and more program code has to be added, in particular in the labs.

This memo begins with a description of the Ethel base system, where comments about the methodological aspects have been inserted at some points. The memo concludes with a separate methodology section that discusses a number of methodology issues in more depth.

2 The Ethel miniworld and simulator

The *Ethel miniworld* contains three main types of objects, namely cities, airplanes, and persons[1]. Each city is assumed to have exactly one airport. At each point in time, each airplane can either be in the airport of one of the cities, or en route from one such airport to another. Furthermore, each person can be in exactly one of the following kinds of state at each point in time:

- In the center of town of one of the cities
- In the airport of one of the cities, but not inside a plane
- Inside one of the planes, which can be either in an airport or en route

Time is structured in terms of discrete timesteps that are informally thought to occur at an interval of around five minutes. At each point in time there is also a current *set of ongoing actions*.

The Ethel simulator is able to represent the current state of this toy world at a particular point in time, and to update the state from one timepoint to the next according to the following procedure. First, it allows external agents (such as the user) or internal agents (such as a planning process of its own) to add actions to the set of ongoing actions. It then considers each ongoing action in turn, in some order, and uses a rule specifying how that action updates the state of the simulated world. These rules also have the possibility of declaring that the action is completed, which means that it shall be taken off the list of ongoing actions and added to the list of completed actions.

We foresee a coming extension of the scenario where the database is extended with more information and where it is used for other purposes besides simulating a sequence of events. These extensions are however not part of the implementation that is used for the present lectures.

3 File structure of the Ethel system

We proceed now to describing the *Ethel system*, that is, the software that is used for demonstrations in lectures and for some of the labs. It must be understood first of all that this system has deliberately been made very simple, so that it shall be possible for students in the course to understand it in its entirety. More realistic systems will of course be much larger, but our demo system still demonstrates important design principles in the real systems.

Since several persons will want to use the system and make their own extensions to it, there is a *base system* that is shared by all and can only be changed by the system administrators, and the *extension* that each of the users has.

[1]We follow common practice in this area and allow us to refer to persons as 'objects'.

3.1 The base system

The Ethel base system is organized as one directory with a few subdirectories, namely:

- `program`
- `database`
- `documentation`
- `runlogs`

Additional subdirectories may be required when new features are added. Files in these directories are used as follows:

- `program`: files containing definitions of Lisp functions and some data items that are used during runs
- `database`: files containing descriptions of data objects of the various types mentioned above: persons, cities, etc.
- `documentation`: text files describing some aspects of the system
- `runlogs`: files containing a log of what has happened during a run of the system.

File extensions are `.cl` except in the documentation section.

The `program` subdirectory contains two files, called `ethel` and `data-access`. The file `ethel` is loaded when the system is started; its job is to provide some introductory definitions and to load all other files that are needed, which means the file `data-access` and all the database files. (This way of organizing the loading of a program is a common one in Lisp).

The `database` directory is described in the next main section.

3.2 The extensions

Each extension contains changes and additions to the base system as made by a particular user. Extensions are independet of each other, and each extension shall be a directory, called the user's *local Ethel directory*, containing the following:

- local variants of the database files in the base system
- additional program files besides the ones in the base system
- other files that the user may wish to put there

The extensions therefore do not use separate subdirectories for program files, database files, etc.

4 Database structure

The database that is given in the Ethel base system is an open-ended one, in the sense that it is intended as a beginning. Students are encouraged to extend the database with additional database objects, additional properties for the objects, and additional object types, and the database is organized in such a way that it shall be easy to make such extensions. The following describes the database of the base system, which can be used for the simulations and as a starting-point for the extensions.

The `database` subdirectory contains files with the following names

- `ontology`
- `airplanes`
- `cities`
- `countries`
- `persons`

The file `database/ontology.cl` contains definitions of the structure of the objects in the other files, corresponding to what one calls 'declarations' in conventional programming languages. Each one of the other files contains the descriptions of a number of objects of the respective types indicated by the name of the file. Thus the file `database/cities.cl` contains descriptions of objects called `stockholm`, `copenhagen`, etc.

The following are the properties that are defined for the objects of the respective types:

For the type `persons`

- `initpos` A descriptor that specifies the initial physical position of the person during simulations. Normally it has the form `(in-city c)` where `c` is a city symbol. In general it can be any of the expressions used for the current state of a person, as described below.

For the type `airplanes`

- `initpos` A descriptor for the initial physical location of the airplane during simulatons. Like for persons, it normally has the form `(in-city c)` where `c` is a city symbol. In general it can be any of the expressions used for the current state of an airplane, as described below.

For the type `cities`

- `distances` an association-list containing the distances from the city at hand to other cities
- `incountry` a symbol for the country where located
- `hasrivers` a list of symbols, for river through the city
- `haslakes` a list of symbols, for lakes bordering the city

- `names` an association-list for names of the city in different languages. Compare the `deflang` command defined below.

For the type `countries`. (Note: this type is not used for the airflight simulation exercise).

- `hascities` list of symbols, for cities in the country
- `capital` symbol for the capital of the country
- `hasrivers` like for `cities` objects
- `haslakes` like for `cities` objects

The base-system action simulator uses only the `initpos` properties. The other properties are used for illustrating the self-describing database (details in section 6). These as well as additional properties may be used in extensions, for example for lab 3.

5 Basic operation of the system

5.1 Starting the system

The system is started from the user's local Ethel directory, so that extensions can be referred to without any subdirectory prefix. It is started by a shell command such as

```
mlisp -L /path-to-ethel-base-system/program/ethel.cl
```

where the program file called `ethel` in the base system is loaded into the Lisp system. This has the effect of loading the program files and data files mentioned above. For the program file `data-access.cl` and the database files, it checks whether the file exists in the local Ethel directory and if so loads it from there, otherwise from the base system.

After the Lisp system has started and the introductory file has been loaded, the system is in standard Lisp dialogue mode (read-eval-print) where the user types in S-expressions to the function `eval`. The following are some functions for the basic operation of the system. In each case, `f` is a filename

- `(ld 'f)` Load a database file in the base system with the name `f`
- `(lp 'f)` Load a program file in the base system with the name `f`
- `(sf 'f)` Save a database file in the extension with the name `f`
- `(lf 'f)` Load a program file or a previously saved database file in the extension with the name `f`.
- `(exit)` Quit the system

For program files, the `lf` command is used for re-loading a program file from the base system after the user has edited it in the local directory, and the `lp` command is used if something has gone wrong and one wishes to re-load the base version of the file.

5.2 The database files

Each file in the database directory contains one single S-expression (parenthesized expression as used in Lisp), with definitions for a number of properties of each one of a number of objects. Here are some comments about methodology that intend to put the structure of these actual files into perspective.

Files such as these are generally designed with two goals in mind:

- They shall be read/write compatible, that is, it shall be possible to read their contents into the program using a 'load' operation (`lf` in our system) and to write the same contents back to the file using a 'save' operation (`sf` in our system). If no changes are done to the data in-between then the new contents of the file shall equal the old contents. In this way, the database files serve to make the data *persistent.*
- They shall be effectively readable for the user, and more specifically the experienced user. It shall be possible to change the data by text-editing the database file and loading it into the running system.

Combining these two goals always requires some compromise: more ease of reading for the user often makes it more complex to read and write the file, and may take more time. In the particular case of the Ethel demo system we have chosen a compromise with little respect for readability by the user, so that the programs for writing and reading the files are trivially simple. This tradeoff is made for the following reasons:

- To allow the users (students in the course) to see the data representation as directly as possible
- To minimize the work for reading and understanding the program
- The convenience of the data notation is a minor issue since only small numbers of objects will be considered anyway.

For larger applications one would probably prefer to use a notation in the files that is more easy to read, since then the trade-offs are different.

5.3 Inspecting and editing the database

The database consists of information about objects in the four types `airplanes`, `cities`, `countries`, and `persons`. The current members of each of these types is kept as the `members` property of the typename, so for example to see the current set of person objects in the database one simply evaluates

```
(get 'persons 'members)
```

The function `show` of one argument shows the current property assignments of the object given as argument.

The fundamental method for changing the properties of objects in the database is using the function `put` (see the list of Lisp functions in the

TDDA23 course materials on the course webpage) or the underlying CommonLisp function `setf`. In programs, properties are of course accessed using the function `get` and changed using `put` or `setf`.

It is also possible to change database contents by text-editing the database file containing the object(s) in question and then loading the file into the system. This is often useful when the properties for many objects are to be changed in a uniform fashion. Notice that if objects are added or removed in this way, it is important to change the `members` property of the type in the file, since it controls what objects will occur in generated ("saved") files.

There are several ways of defining higher-level functions for adding to, or changing database contents; two ways are shown by examples in the base system. First, one may choose for convenience to define his own functions that make a number of property changes to one or more objects so as to fit the needs of a particular application. We exemplify this with the function `deflang` that works as the following example shows:

```
(deflang '(stockholm (fi "Tukholma")(sp "Estoccolmo")))
```

specifies the name for Stockholm in two separate languages, namely Finnish (`fi`) and Spanish (`sp`). Thus, the function shall have one argument, which is a list whose first element is a symbol representing a city, and where the rest of the list is an association-list binding language names to city names, represented as strings. – The exact representation of this information in the database is hidden from the user.

The other method is to define and use a higher-level function that is for general use but can be configured to fit the varying needs of different applications. We exemplify this with the function `def` which takes two arguments both of which are symbols. The first argument shall represent a type; the second argument shall be a member (usually a new member) of that type. The function looks up and uses a function that is attached as a property to the type, and applies it to the member. It is intended that the attached function shall prompt the user for some or all of the properties that are required for objects of the type in question. It may also perform other functions, for example cross-indexing.

The base system contains the definition of the function `def` and the attached, defining functions for several of the types. When using this function for 'city' type objects, please notice that it does not prompt for the `deflang` property, and it has to be assigned separately.

5.4 Modifying the startup

In the simplest case the user will start the base system in the way shown at the beginning of this section, and then load his extension files using the `lf` command. On the other hand, if one should wish that additional program or database files are to be loaded already when the system is started, he should put a copy of the startup file `ethel.cl` in the local Ethel directory, make the required changes in it, and change the startup command accordingly, normally as

```
mlisp -L ethel.cl
```

6 The self-describing database

The files in the `database` subdirectory contains descriptions of objects, in a form that can easily be loaded into the running Ethel system and saved from there. Each file contains the descriptions of a number of concepts that are represented as Lisp symbols; each description consists of assignments for a number of properties of the concept.

When database files are generated or "saved", the information that shall go into the file is stored as separate properties in the Lisp system. The following measures are taken in order to make it possible to generate such files:

- Each file is represented as a symbol, normally chosen as the filename. This *filename symbol* has a property `contents` where the value shall be a list of the objects in the file.
- Each object in a file must have a property `istype`, where the value is a symbol representing the type. Examples of types are `city` and `country`.
- Each type must have a property `props` that is a list of the property-names for all properties that objects in that type can have, and that are to be used when database contents are saved on file.
- The `istype` property of the symbol for a file is `entityfile`. Its `istype` property is `type`, and its `props` property is `(members constants)`.
- A file symbol shall always be included in its own `members` property, normally as the first element in that list.

These conventions, taken together, makes it possible to save a file with a given name, using the stored information about what are the objects in the file and what properties do they have. Notice that if one assigns other properties to a symbol besides those that are listed by the `props` property of its `istype` property, then those properties will not be saved on the file. The information that is stored in the file is such that if the file is written from the system at one time, and read back into a newly started system on another day, then the saved data are still there.

In order for this to be entirely self-describing it is also necessary to save the information about types. This is done in a separate database file called `ontology` which has symbols for types as its members, besides the symbol for itself.

Finally, the following assignments are made:

- The `istype` property of a type symbol is `type`. The `istype` property of `type` is `type`.
- The `props` property of `type` is `(props)`.

In many applications it is also desired to save the values of global variables in database files. The symbols for files therefore also have a property `constants` whose value shall be a list of globally defined constant symbols. The effect of that property is that the current values of the constants are written to the saved file, in such a way that the values are restituted when the file is loaded.

7 Current state and simulation in the base system

The simulation part of the system maintains a current state and contains an operation for proceeding from one current state to the next.

The syntax definitions for S-expressions in this section use the convention that single letters are syntactic variables and are to be instantiated by an actual expression of the kind specified in the surrounding text.

7.1 Representation of the current state

The current state is represented using the following global Lisp variables:

```
*t    current time, represented as an integer
*oa   list of currently ongoing actions
*cp   current plan, that is, current and future actions
*ap   list of airplane positions
*pp   list of person positions
```

In addition there are the following variables that maintain a record of the past simulation:

```
*slog   the past state of the simulated world at successive
        timepoints
*alog   actions that have been performed and already completed
```

The members of the list `*ap` must be lists of two elements, `(a s)` where `a` is the symbol for an airplane, and `s` is an *airplane position* which can have any of the following forms in the base system:

```
(at-airport c)      the airplane is at city c and boarding
                    has not begun
(boarding c n)      where n is 1, 2, or 3: the airplane is at
                    city c and boarding is in process
(ready-to-fly c)    the airplane is at city c and boarding has
                    been completed
```

The second argument for boardingat is a counter that goes from 1 to 3, allowing 3 time units to board a plane.

Person positions are represented in a manner that is similar to airplane positions. The members of the list `*pp` must be lists of two elements, `(p s)` where `p` is the symbol for a person, and `s` is a *person position* which can have any of the following forms in the base system:

```
(incity c)          the person is in the city c and in the
                    center of town
(at-airport c)      the person is at the airport of the city c
(inplane a)         the person is inside the airplane a
```

It is up to the user or programmer to add more types of positions for the lab. The following are some examples of such possible additions:

```
(enroute c c' d v)  the airplane is flying between the two
                    cities c and c', d is the distance that
                    has been travelled so far, and v is the
```

```
                    current velocity
(disembarking c n)  analogous to 'boarding'
```

7.2 Time, update, and simulation

The current state is calculated at successive timepoints that are represented as numbers 0,1,2,3... The Ethel state update function is the driving routine for simulations. It can be called when the global variables for current state model the microworld for a particular timepoint. It updates the values of those variables so as to progress to the next timepoint, and it does so by considering all the ongoing actions, that is, all the elements in the list `*oa` and performing the changes in the simulated world that are required by each action in turn.

One important issue in such systems is whether the outcome of the state update function depends on the order in which the actions are 'visited', and if so whether that is consistent with the model of the world being used. We do not address that issue here.

We shall now first define actions, and then the state update function.

8 Actions

8.1 Terminology and notation

We start with definitions of concepts and notation in general, not only for the Ethel base system.

The following three concepts are related but distinct: *action verb*, *action expression*, and *action*. An action verb is a symbol that specifies one type of action, represented as a symbol for example `let-board` or `fly-to`. An action expression is a list where the first element is an action verb, and the other elements of the list are arguments to the action verb. The following are some examples of action expressions:

```
(board-plane peter lh-12)
(fly-to lh-12 london)
```

The intended meaning of an action expression is up to the system designer, but conceivably the action expression `(board-plane peter lh-12)` could be used to mean "Peter boards airplane `lh-12`", and the action expression `(fly-to lh-12 london)` could be used to mean "airplane `lh-12` flies from its present position to London".

An *action* is a specific occurrence; we only consider those actions that can be characterized by action expressions. For example, if Peter boards airplane lh-12 twice, on two different days, then those are two different actions but they are characterized by the same action expression.

For modelling purposes we make the assumption that every action is characterized by a starting time and an action expression, and that those two specify the action unambiguously. None of the following is therefore considered to be possible:

- An action without an action expression
- An action with more than one action expression
- An action without a starting time, or with several starting times
- Two separate actions that have the same action expression and the same starting time

At a particular point in time during the development of a system, for example as simulated by Ethel, the following kinds of actions are of interest:

- Ongoing actions, for which there is a starting time that is less than or equal to the current time, but for which the ending time has not yet been obtained (and is going to be strictly greater than the current time)
- Past actions, for which there is a starting time and an ending time, both of which are less than or equal to the current time
- Planned actions, that have not yet started (and it is not sure that they will ever start)

These characteristics with respect to starting and ending time apply *between* state updates, but clearly not in the middle of a state update. In other words, they are part of the *invariant* with respect to the state-update operation.

For planned actions it is of interest to reason about their (future) starting time and ending time, for example for specifying that one planned action shall occur after another one. Similarly for ongoing actions one may wish to reason about their ending times.

Ongoing actions are written as `(t e)` where `t` is an integer representing a timepoint and `e` is an action expression. Past actions are written as `(t e t')` which is like for ongoing actions except that `t'` is an integer for the ending time. Planned actions are also written as `(t e t')` but here `t` and `t'` are symbols such as `t0`, `t1` etc that represent the future starting and ending times. More about this in section 10.2 and in later lectures in the course.

8.2 Defining the behavior and properties of actions

Each action verb has one or more attached functions which specify some aspect of the behavior and properties of actions with that action verb (that is, actions whose action expression begins with that action verb). One attached function is required, namely the function in the property `update-fn`. This is a function that takes an action-expression as an argument and performs those updates of the current state (in the simulation, for example) that are required by that action. The state-update function in the Ethel base system uses the `update-fn` property of the verbs in ongoing actions.

A number of other attached functions are also of interest and should be present in full-scale systems of this kind, in particular:

- An *applicability condition* which checks whether a particular function can be started in the current state

- A *syntactic correctness check*, i.e. a function that checks whether a given action expression is correctly formed with the right number and right kinds of arguments, etc.
- A *linguistic expression function* which finds an adequate sequence of words for describing the action in a given language

The Ethel base system contains the beginnings of an implementation of the applicability condition, but nothing for the other two. They are left for future labs and other future work.

8.3 Action definitions in the base system and in extensions

The base system only contains the definitions for one single action verb, namely the action verb `let-board` (previously called `start-boarding` in the lectures). As long as one has only loaded the base system and no extensions, `let-board` is therefore the only action verb that can be used. The assignment in lab 3, in particular, is to make the definitions for additional kinds of actions.

9 The state update function

9.1 The four versions of the state update function

The program file `data-access` contains four different definitions of the update function and of the associated rule for performing one step in the action `let-board`. These definitions illustrate different programming styles and programming techniques in Lisp.

Three techniques are illustrated in these program versions:

- The use of auxiliary functions to obtain a certain degree of data abstraction, so that programs can be less dependent on the actual representation of data in data structures
- The use of attached functions or procedures which are the properties of symbols that occur in the database
- The use of an embedded, special-purpose language, in this case a simple rule language, on order to obtain a representation for the state update functions that is closer to how the designer thinks about them.

Version 1 is a definition of the state update function that contains the procedure for the action `let-board` within it, using a `case` expression, and without any use of data abstraction.

Version 2 is similar in structure to version 1, but is written with data abstraction using a number of small, auxiliary functions.

In version 3, the case expression from versions 1 and 2 has been replaced by the use of attached procedures for the action verbs. We show the attached procedure both without and with data abstraction.

Version 4 differs from the previous versions in that the rules for the action verb are written in terms of an embedded rule language, whereas the first three versions define each action by a piece of program.

The state update functions for these versions are called `update-state1`, `update-state2`, and so on. In all cases they are called without argument, for example `(update-state1)`. If a shorter form is desired one may e.g. define `ups` as

```
(defun ups ()(update-state2))
```

Ad hoc definition and use of auxiliary functions such as this one is standard practice when working in an interactive and incremental computing environment such as the one in Lisp.

The reader is strongly recommended to study the actual code in these alternative definitions and to make sure that he understands how they work, before proceeding to the next subsection.

9.2 Discussion

The following are some comments comparing the versions of the state update function.

Case expressions vs attached procedures. In symbolic computation it happens quite often that the program has to consider a number of different cases by different procedures. Both case expressions and attached functions or procedures are effective ways for structuring such programs. Case expressions are usually to be preferred when there are only a few cases and the procedure for each of them is short. Attached procedures usually provide better modularity and legibility when there are many cases and the procedure for each of them is lengthy. In particular, they make it possible to place different attached functions in different files, so that they are in a context where they belong naturally.

Attached procedures are also favored when one wishes to operate automatically on the program, for example by generating some of the cases automatically. It is easier to generate and analyse a property for an object than to insert additional cases inside a larger program.

The distinction between these alternatives is not altogether sharp, since the modularity advantage of attached procedures can also be achieved by ordinary procedures that are systematically organized and named, for example `let-board--update-fn`, in combination with case expressions only containing calls to those procedures in the obvious way.

The use of embedded, special-purpose languages. This is an important technique which was illustrated here for the rule language. Special-purpose languages are useful when one faces a number of separate programming tasks that have a general character in common but considerable variation within that framework. The definitions of the state update for a number of different action verbs is an example in point.

The pros and cons of data abstraction. The use of data abstraction is an important principle in general software engineering[2], and the reader may

[2]The concept of data abstraction is importnat, but it is sometimes omitted in

have been surprised when studying version 1 of the state update function – why not go directly to version 2 where data abstraction is being used?

The answer is that although data abstraction is of course often used in Lisp programs as well, and there are also other ways of achieving it besides the simple approach in version 2, still the matter is not as simple as one might think. The following aspects are also part of the picture:

1. The availability of a general-purpose textual representation of data as S-expressions (parenthesized expressions) in Lisp allows the programmer to understand important parts of his program in terms of examples of S-expressions. This possibility does not exist in conventional languages, unless the programmer introduces it himself, which means that Lisp has another tradeoff between the use of concreteness and generality.
2. The introduction and use of an embedded language, such as the rule language, is another way of abstracting away not only the actual data structures, but also the skeleton of the control flow. In version 4 the specification of an action verb does not need to use any access or update functions at all for the data structures. One may still consider whether data abstraction is useful when writing the interpreter for the embedded language, but then one deals with a small piece of code where there are also advantages with staying quite close to the actual data representation.

Notice also our use of the state update function in version 4 as an example in the lecture on partial evaluation. When working that example it was an advantage to use definitions that were expressed directly in terms of elementary functions such as `car` and `cdr`.

Error checking and robustness The implementations in the Ethel base system contain very little checking for correctness, besides what is provided by the dynamic type checking in the Lisp system itself. Checking for errors is still very important, of course[3], in order to guarantee the robustness of a piece of software that goes into practical use, but this doctrine is also somewhat modified in the Lisp tradition, for example for the following reasons:

1. For those checks that have to be done at run time[4] the simplest way of performing such checks is to integrate them in the program at the points where the checks need to be done. However, this tends to clobber the program and make it difficult to see the main lines in it. It is therefore desirable to organize the program so that these checks are separated out as much as possible, and there are several ways of doing this in Lisp. The introduction of the action applicability test through the separate `precond-fn` property in the Ethel base system is one example.
2. Some of the reasons for static checking in conventional languages are made unnecessary by the storage management techniques in Lisp,

elementary computer science courses. If you do not know what this is about, then skip this subsubsection.

[3] If you are not familiar with what this is about, then skip this subsubsection.

[4] For example, the check that the request by a user to start executing a particular action is a correct one.

such as garbage collection, typed pointers and dynamic type checking. (These implementation techniques will be presented in a separate lecture note).

3. When embedded languages are used, the checking aspects of their interpreters have to be designed by the programmer anyway since the type systems of standard programming languages can not do the job. In such situations it is natural that the design and use of correctness checking is located in higher layers of the software hierarchy and are considered the responsibility of the programmer.

4. It is always true that testing a program by running test examples is never guaranteed to find all bugs. It is equally true that the use of test examples is very important in practice as *one* method for checking out a program. Furthermore, the use of test examples tends to be more effective for symbolic computation than e.g. for core systems software (operating systems and the like) or large numerical calculations – errors do not hide as easily in symbolically oriented programs.

9.3 Logging

The `sf` command for saving a database file is used for the purpose of logging as well. Notice that the global variables `*slog` and `*alog` contain lists of past system states at successive timepoints, and of past, completed actions. The function call `(stafil n)` where n is a symbol, defines `n` to be the name of a file that can be saved using the function `sf`. This is useful to saving past history so that one can inspect it off-line and see what has happened. It is also used for labs, where this provides a way of saving the log of a run so that it can be turned in for the lab report.

The definition of the function `stafil` is in the program file called `data-access`, and is quite short. it is recommended to take a look at it and figure out how it works; it is an example of how a self-describing database, such as even the simple `sf` and `lf` functions in Ethel, can be used for a variety of purposes.

10 Recommended extensions

This section mentions some possible directions for extension of the Ethel system. They have not been implemented in the base system. (This section is incomplete, and additions may be issued later).

10.1 Distance, velocity, and flying time

We recommend that the following model shall be used for lab 3. Distance is represented in kilometers, for example 1600 for a typical flight. The `distance` properties of cities is used for representing distance between them. The third argument of the airplane location operator `enroute` is used for showing the distance that has been traversed so far during the flight, and as the flight proceeds it goes up from 0 to the distance between the first and second argument. The fourth argument represents the “velocity” of the airplane as an integer between 0 and 90 in steps of 10, indicating the number of kilometers flown during one time unit. Recall that the time unit is roughly 5 minutes of flying time.

10.2 Syntax extensions for actions

It is convenient to allow action expressions to have optional arguments, for example

```
(fly-to lh-12 london :marching-altitude 8500)
```

In some cases one wishes to assign properties to action expressions or to actions. One practical way of doing this is to generate a unique symbol for the action and to refer to it in an optional argument, for example

```
(fly-to lh-12 london :marching-altitude 8500 :symbol fly-to-23)
```

where `fly-to-23` is a symbol that has been generated in such a way that one knows it has not been used before. That symbol can then be given an `istype` property and all the properties that can come with the type.

10.3 Plans

A plan is a list of actions using symbolic timepoints t0, t1, t2, ... By convention t0 is the starting time for the plan. For example,

```
((action t0 (letboard lh-12) t1)
 (action t1 (flyto lh-12 london) t2)
 (action t2 (let-disembark lh-12) t3) )
```

is the plan where the airplane `lh-12` allows passengers to board, then flies to London, then allows them to disembark.