Provisions for flexibility in the Linköping office information system (LOIS)*

by ERIK SANDEWALL, GÖRAN HEKTOR, ANDERS STRÖM, CLAES STRÖMBERG, OLA STRÖMFORS, HENRIK SÖRENSEN and JAAK URMI

Linköping University
Linköping, Sweden

1. CHARACTERISTIC PROPERTIES OF THE LOIS SYSTEM

The Linköping Office Information System (LOIS) is an integrated system of facilities for text preparation, data base management, communication by computer, and miscellaneous other services. It is an experimental research system, which is used by researchers and secretaries in our own research group.

A significant consideration in the design of this system was how to provide very large flexibility, so that each user could have his or her customized variant of the system, without imposing an unrealistic burden of programming on either the users or a system group. Two complementary ways were recognized for achieving that flexibility:

- *Adaptation by the user:* the system could include novel facilities which, like modelling clay, allow the user to adapt them to fit his/her needs;
- *Application development tools:* there could be tools which enable a trained person to tailor facilities very easily, for individual users or groups of users.

Such tools should be easy to use, so that only moderate training is necessary, but there is no requirement that every user should be able to use them.

Both of these approaches have their merit, and both have been used in the LOIS system. The advantage of adaptation by the user should be clear; and application development tools are appropriate not only for harder tasks, but also for facilities which involve several users, i.e., what we shall call information-flow facilities below.

Another aspect of flexibility was that the standard services in the system should be easily interfaceable, so that they can be run together. This is a demand on the programming techniques that are used in building the LOIS system.

A second major purpose in building the system has been to experiment with unconventional terminal equipment. In particular, we have set up a low-cost device for output using arbitrary fonts (boldface, italic, larger fonts for headlines, foreign alphabets, etc.) using a high-resolution electrostatic plotter, and built software for supporting that medium.

The following sections will describe these various aspects of the system in more detail. Thus the intended purpose of the paper is *not* to discuss the general-purpose facilities in office information systems. A number of significant and well-known system development efforts (for example at Stanford Research Institute, IBM Research Centers, and the University of Pennsylvania (Ref. 1)) have set the standard for such systems.

Before we proceed, we should however give a short summary of the services in the system, as seen from the individual user. LOIS recognizes three major ways of structuring information:

text, i.e. ordinary, continuous text in natural language (English, Swedish, etc.);
data, where information is organized as a table and/or as a form containing different fields or slots which may contain items of information;
notes, which is an intermediate form between text and data. A note is a short text which is associated with additional information organized as a *data* record. In practical usage, a note may be a message sent from one user to one or more other users, containing the text of the message plus information about sender, receiver(s), date, topic, etc. In another usage, the *note* may be one person's notes about the contents of a book, with associated information about author, title, and classification.

From another perspective, users will recognize some facilities as *local*, i.e., only one user is involved when they are used, for example for personal data bases, and other facilities which are *shared*, i.e., they involve the user in communication with other users, for example computer mail.

2. TEXT PROCESSING FACILITIES FOR FONTS

The LOIS system uses the common strategy of having general-purpose text editors and separate formatting ("run-

* This research has been sponsored by the Swedish Board of Technical Development (STU) under contracts Dnr 77-4420, 77-4380b, and 78-4165.

off") programs. Besides supporting some conventional output media such as a Diablo printer, it also has a font printout system based on a Versatec electrostatic plotter. This system is able to produce output that approximates ordinary printing, with facilities for several *fonts*, such as italic font (cursive), **larger** and bold-face letters for major headings, etc. Different characters in a font may have different widths. It is possible to define fonts for other alphabets or for special signs, and use them freely mixed with the regular text. In particular, mathematical text as well as many kinds of figures may be produced in this fashion.

However, the graphic quality of this system is not fully commensurate with regular printing using typesetting. In particular, *italic* letters often appear a little unsteady if you look at them closely. We still believe that this quality is sufficient for many purposes. When compared to a phototypesetter, this equipment has the disadvantage of lower graphic quality, but also several advantages:

—the system is cheap enough that you can afford to have it within easy reach of each user;
—fonts may be created or modified at the site;
—the same device may also be used for vector plotting and grey-scale pictures (facsimile).

2. Text processing facilities for fonts.

The LOIS system uses the common strategy of having general-purpose text editors and separate formatting ("runoff") programs. Besides supporting some conventional output media such as a Diablo printer, it also has a font printout system based on a Versatec electrostatic plotter. This system is able to produce output that approximates ordinary printing, with facilities for several fonts, such as italic font (cursive), **larger** and bold-face letters for major headings, etc. Different characters in a font may have different width. It is possible to define fonts for other alphabets or for special signs, and use them freely mixed with the regular text. In particular, mathematical text as well as many kinds of figures may be produced in this fashion.

Figure 1—Sample printout from the Font System.

Technically, this system consists of a domestic LYS-16 16-bit small computer (soon to be replaced by an LSI-11 computer) combined with a Versatec graphic printer. The Versatec is an electrostatic raster printer with a resolution of 200 points per inch (8 points per millimeter). The LYS-16 contains software which will accept bit-pattern definitions of the characters in one or more fonts, and print a given file using these bit patterns for each character. The combined LYS-16 plus Versatec system may be viewed as an "intelligent printer."

The font system has been modeled on a similar system at the MIT and Stanford Artificial Intelligence Laboratories, which however use a Xerox Graphic Printer (XGP) instead of the Versatec printer. The resolution is almost exactly the same. On comparison, our system seems to give less contrast, but also less noise, and prints at a lower speed (probably mostly due to a slower processor).

Formatting

The formatting for the font printout system is done using CRAWL, a locally built formatter which besides the support of fonts, also has a number of other non-standard facilities:

>automatic hyphenation (more necessary for Swedish text than for English text since Swedish makes frequent use of long, composite words—like German);
>the formatter co-exists with a Lisp programming system, which means that commands in the source text can call arbitrary Lisp functions for specific purposes. This gives the same advantages as having a macro facility in the formatter (as is used in, e.g., the Unix system, Ref. 2), but with the significant difference that a full programming language provides services such as data base access, availability of a program library, and easy interface with other programs, which a macro system which only serves the formatter cannot be expected to provide.

A text formatter embedded in the programming language SAIL at the Stanford Artificial Intelligence Laboratory, offers similar advantages.

Preparation of fonts

In the font printout system, each letter in each font is defined by a pattern of many small points. An ordinary small letter in a common font is about 15 points high, for example. The definitions of the point patterns of the characters in all fonts are stored on the central DEC-20 computer, and sent to the font printout device when needed.

The work of building up new fonts may require a considerable effort. Through the generosity of the M.I.T. A. I. group, we have a copy of their fairly large font library, which could be used after a routine shift of representation. However, we also have a need to modify old fonts (for example to create the Swedish letters with diacritics), and to create entirely new fonts for specific purposes.

Two tools have been built for these purposes, a *font editor* and a *font generator*.

The font editor

The font editor is a tool for defining and changing the point-by-point definition of fonts. The font editor is in itself a program, but it requires a specialized terminal, which has been built by the Electrical Engineering department at our university. The system allows the user to edit one character at a time, and to view the character in two versions on the terminal's display screen, namely both a blown-up version where each point is clearly discernible, and a realistic version which looks like and gives the same impression as the character will have on paper (only magnified by about a factor of two). The editor allows the user to add and delete individual points or rows of points by hitting keys on the keyboard, and to see the effects of each change immediately.

The font editor has been very useful, both for modifying MIT fonts to contain Swedish characters, and for building up fonts of, e.g., mathematical symbols.

The font generator

Although the font editor greatly facilitates the task of building up a font, doing so still requires a lot of work. Sometimes it is routine work, namely if letters of the same general shape are desired in several different versions, with different height, different boldness, roman or italic, with or without serif, etc. For such situations, we have developed a *font generator*, which generates fonts automatically from given specifications.

The font generating program takes two kinds of inputs. One input is the desired specification for the new font, i.e., values for the desired height of big and small letters, a measure of the desired boldness, etc. This input is specified anew each time the program is run.

The other input consists of structural *descriptions* of the characters in an alphabet, saying, e.g., that a capital "L" is a vertical line with a shorter horizontal line extending to the right from the base of the vertical line. These descriptions are expressed in a formal language, and are semi-constant, in the sense that the description of the Latin alphabet can be used repeatedly for different dimensions, but also if some other symbol set is desired (such as mathematical symbols) it is well defined how to write the structural descriptions of them also.

A program for the same purpose written by Knuth at Stanford uses mathematical functions (splines) to describe the curvature of the letters. Our system builds up letters from pre-defined segments, which can be designed by a combination of manual design and automatic generation. This is particularly useful for bit-matrix output devices whose resolution is almost discernible for the eye, since the effects of direct discretization of continuous functions may then be disturbing.

In addition, there are a number of smaller service programs for operating on fonts, such as a program for rotating the characters in a font by 90 degrees, and a program for rotating each page in a text file correspondingly.

3. STRUCTURED DATA FACILITIES

A significant part of the routines in an office environment deal with structured data rather than free text. The structured data facilities in the LOIS system, which aim to support this need, are organized around a screen-oriented *data editor* called IFORM. This system allows the user to view structured data on his display screen, organized into *forms*, i.e., fixed layouts containing certain fixed *text fields* and other fields, *data fields*, which can be filled with the desired data. Just like a text editor is used both for entering text and for changing existing texts, the IFORM data editor is used both for entering, viewing, and changing structured data.

Typical uses of the data editor in an office environment may be to maintain an address register, a register of reports and memoranda, a register of allocation of offices, or a register of equipment used in the group.

The basic idea in the IFORM system is of course available also in some commercial systems on the market, but IFORM contains some facilities which are not usually found, in particular:

- >*programmability:* each data field may be associated with procedures in a number of different "slots" for defining specialized rules about how to interpret input into a field, check restrictions on the proposed input, print out the contents of a field on the screen, obtain consequences (side-effects) from new values, etc.
- >*tables within a form:* a form may contain a table which consists of a number of occurrences of a sub-record. This is useful for example when the form for a person contains a table of the trips he has made during the year, indicating the date, purpose, and destination of each, displayed with one line for each trip and one column for each fi Single-key editor commands allow manipulation of these sub-records, e.g., insertion and deletion of sub-records in the sequence.

To support this data handling facility, there are a number of other tools, in particular:

Data base with exchangeable access methods

The forms supported by IFORM are a standard interface for the user, through which he or she can access a number of different data bases, potentially even on several computers of different kinds. (This is in accordance with the proposals of the CODASYL End User Facility task group, and this idea has been articulated and extended within our cjllaboratory by Erland Jungert). IFORM is therefore organized so that access to the data base goes through a number of access routines associated with an *access method*. Additional access methods may relatively easily be added.

The ability to exchange access methods for the data base is in fact useful for two reasons:

- >for interfacing IFORM to a new data base:
- >for using one access method during development of an application and in its prototype stage, and another access method during production use of the same system.

The layout editor

IFORM uses a *form description*, i.e., a structure which describes the desired layout on the screen: which fields are used, what are their X-Y coordinates, etc. The *layout editor* is an interactive tool for building up and modifying such forms.

4. NOTES AND COMMUNICATION

The third information structure in the LOIS system, *notes*, are objects which consist of a short segment of free text, combined with a number of *properties*, each of which is a keyword and a corresponding value. The following is an

example of a note which a user may have during or right after a telephone call:

TALKWITH: Larsson
DATE: 1978-10-24
TOPIC: Holiday season, Vacations, Production
TEXT: Unusually many people are using remaining vacation days for extra vacation around Christmas. Production of bicycle chains will be particularly delayed.

The following is an example of a note which describes a computer terminal used in a research group:

TYPE: Hackmatic 1521
INVENTORY-NUMBER: 410
LOCATION: NB-156
CONNECTED-TO: DEC-20, PDP-11C
TEXT: This unit has required repeated service with various faults and seems to be flaky. Erasure of one line at a time does not work and seems to be permanently unfixable.

The POST subsystem in LOIS maintains for each user a database of notes, and enables the user to retrieve notes with given properties, to add new notes, to modify the properties of existing notes, and to call an ordinary text editor for modifying the textual content of a note. This information structure can be utilized for a number of different purposes, as suggested by the examples.

The POST system should really be viewed as a data base system which is able to also contain textual objects. It already provides non-trivial search facilities in this data base, and interfaces to other data base handling facilities, such as IFORM in the LOIS system, seems straightforward.

The present POST system encourages the texts to be short, but it is a straightforward extension to also allow notes whose text parts are conventional, larger text files for manuscripts. A system like POST might then be used as a more powerful substitute for the conventional file directory, and would allow the user to store arbitrary information about his files in the POST data base. This design would also give the user full data base capabilities for administering his "directory."

One particular use of notes is for *communication* between users, where each message is well expressed as a note, with properties indicating the names of sender(s) and receiver(s) of the message, the date the message was sent, the topic and other classification of the message, etc., and where the essential content of the message is conveyed in the free-text part, at least for simple messages. The POST system includes a message-passing facility, so that each user can send and receive messages, and the general-purpose data base facilities of POST can be used for administering incoming and outgoing mail.

Notice that the properties associated with the note are not only used for "system" purposes in the mail system, such as administrating the names of sender and receiver. They are also used by the sender and the receiver for representing information which classifies or otherwise describes the content, purpose, or use of the message. In particular, the receiver may change the values of properties, or add new properties, to messages that he has received. Also, it is sometimes very useful to represent some or all of the contents of the transferred message as values of properties, rather than in the free-text section.

One example of the use of such *structured messages* is the following: a message about a seminar may represent the name of the lecturer, the topic, the date, time, and location as separate properties. This greatly facilitates interfacing the message sending system to other facilities, such as a computer based calendar, or a system for generating summaries of recent activities.

The idea to base a computer mail facility on a data base handler for information organized as notes, appears to be a very powe[illegible]ne. It provides a good basis for other communicative facilities, which may be more structured than simple mail sending, for example a *computer conferencing* system (which we have programmed but not yet put in operation), or for computer based decision making.

As the name indicates, the POST system started as a mail system, and its usefulness for storing one user's private information was recognized and exploited only gradually. The ability to organize one's personal information as a large collection of notes, and to have a full data base facility for keeping the notes organized, are only starting to be exploited, and we believe that several additional uses of this structure will be found as the system is used.

5. APPLICATION DEVELOPMENT TOOLS FOR INFORMATION-FLOW FACILITIES

The office environment contains many routines where a "packet" of information circulates between several "stations". For example, a purchase order is initiated by one person, and passes stations for approval, for selecting the vendor, for receiving and checking the goods, and for paying the bill. Each such application can be characterized as a flow of information packets, which follow certain paths; which sometimes are delayed awaiting some external event; which accumulate and give off information during their path through the organization; and which require human intervention at many of the stations.

As seen from the human user, these information flows are used for routine communication within the organization. In paper-based communication, one often prefers to use forms for this purpose, and in a computer-based system one would also desire fixed layouts (forms) rather than the free format of computer mail. For information flow with very high volume, for example in banks, this has of course been realized since a long time, but we are concerned with tools for low-volume information flow which must be supported locally.

Each information-flow application will involve several users, and symmetrically, one user will often be involved with several different information flows. In a hospital for example, the head nurse of a ward will be involved with at

least the following flows:

>patient registers, undergoes treatment, and leaves;
>"purchase" orders for laboratory analyses for patients in the ward;
>scheduling of working hours for different categories of personel in the ward;

and so forth. The entire office information system should therefore have a matrix structure with "users" in one dimension and "information-flow applications" in the other.

There is a significant structural difference between *development time* and *usage time*, then. When the system is used, each user wants to have his system as an entity, and to be able to switch easily between his part of each of the applications. In particular, he wants to be able to transfer data easily between the messages in different information flows. But when an application is developed, it is essential that all the work stations for that application are developed together.

Such information-flow applications are supported in the LOIS system by a combination of two measures. First, the software in the usage-time systems that are run by the individual users, have a well-defined structure so that additional facilities can be inserted automatically. Second, the LOIS system includes a modelling language and an application development tool which allow its user to build a description of an information-flow application in problem-oriented terms, and generate the appropriate contributions to the relevant usage-time system automatically.

The description of an application consists of three parts:

>a description of the information flow as such, showing the successive operations (initialization, additional data entry, delay, copying, etc.) which happen along the way;
>a record declaration which describes the structure of the information packets that travel in the flow;
>a form description which defines the appearance of this record on screens and paper. This description is entered and maintained using the IFORM sub-system that was described above.

In addition, there is one master description of the organizational structure, which is used as a common reference by all information-flow models, and which relates them to the usage-time systems.

This application development tool is somewhat interesting from the point of view of programming methodology: usually a programming system handles entities ("programs") which contain the specification, or a part of the specification, for *one* executing process in the computer system. In our case, the application development system contains specifications for a set of coordinated processes, which are to be run by different users and often at different times, and which are all generated from the application description.

A more detailed description of this system has been given in Ref. 3. A system with some similarities has been developed by Hammer et al. (Ref. 4).

6. DIRECTORY SERVICES

Many parts of the LOIS system require that the system maintains directory information, i.e., information about information stored in the system. Examples of directory information are:

>catalogues of the text files and data files maintained in the system;
>classification information for notes;
>structure descriptions ("declarations") for the data files maintained using IFORM, including information about the intended content of each data field.

In addition there is directory information which is essential for the proper functioning of the system, but which is or at least should be invisible to the user, such as:

>information about the different versions of a text file which appear in the course of successive operations (formatting, transcription to another alphabet, transformation to the printout conventions of a particular output device, etc.);
>information about the access method used for a data file maintained by IFORM's data facility.

One basic design decision in LOIS has been that all such directory information should be maintained *in the data base of the system*, so that it can be accessed and used by the standard software facilities in the system, and by a gradually growing set of application programs. At present the following services are provided:

>classification of data entities in an application-oriented hierarchical system, so that entities may be classified, e.g., with respect to what part of the owner's responsibilities they are used for. Such a structure is necessary when the number of text files and data files in the system increases: simple mnemonic naming of each file individually is not sufficient for structuring this body of information;
>documentation of program modules, user systems, etc.;
>automatically performing certain routine operations on text files, such as formatting and similar transformations before printout. This facility is viewed as a first step toward a system which "knows" about what routine data processing is needed in the application environment, and performs the appropriate operations at appropriate times. There are many similarities between this concept, and the modelling of information flow *between* users described in the previous section.

7. ARCHITECTURAL CONSIDERATIONS FOR FLEXIBILITY

New users are introduced to the LOIS system by learning about the basic facilities, for operating on texts, structured

data, and notes. But these sub-systems may be modified and recombined in many ways, and we expect that such modifications shall be done each time the system is used in a new environment or for a new class of tasks. It is not intended that every user should be able to modify the system, but it *is* intended that modifications can be done very close to the environment where they are going to be used, and preferably by one user of the system.

This flexibility of the system has been exercised to some extent within our environment, although additional experiments remain to be done. Several programming techniques are used to achieve flexibility and adaptability:

Use of a residential programming system

A residential programming system can be viewed as a data base system which is able to contain programs in its data base, and which contains an interpreter for programs that are stored there. Such systems provide unusual possibilities for program structuring, since programs and data can be integrated. This is useful for example for all programs that decode a repertoire of commands, and take appropriate action for each of the commands. There are many examples of such programs in office applications, for example editors and formatting programs for free text.

Another advantage of residential programming systems is that programs can be gradually modified and extended, even during an interactive session. This makes it easier to maintain a system as a collection of modules, which are loaded when needed.

Rich parameter structures

Several of the programs are directed by parameters which are represented as LISP list structures, which allows a rich and easily manipulated parameter language. Examples of use:

>the IFORM data editor is parameterized with respect to layouts. The layout description specifies the location, content, etc., of each field. The non-trivial facilities in IFORM, such as for supporting embedded sequences of sub-records, depend strongly on this parameter structure;

>the character description language used by the font generator, DRAW, is an example of a rich parameter language.

The use of a residential programming system facilitates the use of rich parameter structures, since programs and parameters are stored in an integrated fashion in the programming system's data base.

Superroutines, i.e., programs with handles

Parameter structures are usually set up so that the parameters and/or the object data may contain the names of LISP functions, which are called when the data are processed. This technique assumes of course equivalence between programs and data. Some examples of its use are:

>the layout descriptions used by IFORM contain handles where calls to arbitrary (LISP) functions may be inserted, for specifying specialized printout formats, read-in functions, checking functions, or other aspects of the system's processing;

>the POST sub-system allows messages and other notes to have a property which names a (LISP) function which is called when the note is processed. In this fashion it is possible to arrange that messages are processed automatically on reception, without need for manual intervention by the nominal receiver of the message. For exampl ıser may send out a query to a group of other users, where each query requires the recipient to answer a number of questions (represented as properties) and return the questionnaire, and where the initiator may set up a program which receives and summarizes the returns.

>the CRAWL text formatter is designed so that the source file may contain calls to arbitrary (LISP) programs, which are executed when the call is encountered, and which, e.g., may generate a part of the desired printout (e.g., may make data base access and generate a table of structured data).

Extendible command sets

Several of the sub-systems contain specialized command languages, either for interactive use or for use in source files (in CRAWL). Usually they have been set up so that additional commands can be defined as LISP code in a modular fashion, and so that definitions for additional commands may be loaded into a sub-system even in the course of an interactive session. This technique makes it possible to keep the basic system small and simple. Instead of proliferating it with a large repertoire of special-purpose commands, the specialized commands are kept as separate modules and loaded into the system when needed.

>the IFORM data editor may be extended with new commands which are specialized for application-oriented situations. For example, if IFORM is used to maintain information about patients in a hospital ward, one may have specific commands which are used when a patient enters or leaves the ward, and which initiate the operations (such as transfer of information to and from an archive) which are required at this event;

>the layout editor which supports IFORM may similarly be extended with specialized commands, for example for introducing new kinds of fields. As one example, when the IFORM system was adapted to supporting VIEWDATA terminals, special commands were defined for inserting color shifts into the layout description.

Message passing between programming systems

For each user, or group of users with similar needs, there is a version of the residential programming system which has been loaded with the programs, parameter structures, and other data which that user needs. Orthogonally to this set of *user systems*, there is also a set of *development systems*, namely one for each information-flow application, and one for each general facility (such as the formatter). The contributions which are made from development systems to user systems are transferred by a kind of message passing. The "systems" in this sense are therefore viewed as independent entities with local autonomy.

Combinability

Another characteristic property of the system is that different modules can be made to interface with each other, using either "subroutine" calls or data transfer as the interface. This property of the system is made possible by a combination of two circumstances, namely (1) the flexibility properties which have just been described, and (2) the "callability" properties through all levels of software in the system we are using. This latter property is based on the TOPS-20 operating system, which for example makes it easy to let one process call another process recursively, including the operating system; but it is also due to the Interlisp system, which forwards these properties of the operating system to the programmer on the Lisp level.

The callability property has of course also been followed up within the LOIS system itself, where various sub-systems have been set up so that they can be operated both by direct user commands during an interaction, and as subroutines which are called from other programs.

Some examples of this combinability property in LOIS are:

>the note handling system may call the text editors recursively, for operating on the textual content of a note. The same applies for the data editor;
>the data editor has been equipped with a command which generates messages automatically using information in the data base, and calls POST for having them sent out to recipients. This is useful, e.g., for sending out reminders automatically according to criteria in the data base, such as a reminder to return a borrowed book when the time is out;
>the text formatter CRAWL goes into a dialogue with the user when a syntax error in the input file is detected, allows the user to correct the error, and then proceeds through the same source file with no need to start over from the beginning of the file;
>through the ability to define reception procedures for messages, it becomes possible to arrange that the contents of structured, incoming messages are gradually accumulated to the data base, where they can later be inspected using POST or IFORM

Additional services

A few other programs have been written besides the basic facilities and their derivatives, in particular:

>a personal calendar, with facilities for displaying and editing the current state of the calendar, and for booking a common meeting-time for several users of the system;
>a personal agenda, i.e., a program which maintains a structured list of assignments that the user intends to perform, and provides support for editing this agenda.

8. IMPLEMENTATION TECHNIQUES AND EXPE NCE

The LOIS system has throughout been intended as an experimental system, developed as a research project. The system has been designed so that it could be used within the group (for testing and for feedback on the design) but has not been intended for wider use. We therefore assigned high priority on the ability to modify and extend the system in the course of the project. For these reasons, and since we had access to a sufficiently large and powerful computer, we made the essential design decisions to let most part of the system operate on the DEC-20, and to write most parts of the system using the programming system INTERLISP. (Remaining parts have been written in assembler or Simula.)

At the same time, we also wanted to distribute some of the functions in the systems to separate and smaller processors. The locally built LYS-16 computer was used for this purpose.

In this final section, some aspects of this software strategy will be discussed.

Workspace systems vs. conventional systems

Traditional computer programs operate with one or more files as input, similarly for output, and perhaps some interaction with a user. However, the INTERLISP system (like other LISP systems, and like APL systems) are organized so that the user will conduct an interactive session talking to a *system* which maintains a *workspace* for the duration of the session. This property is very significant for debugging and general maintenance of programs. It does not have to be used for the application situation, since one can write LISP functions which have the traditional file-in, file-out organization, but it is possible to use it for the application situation as well.

In LOIS, both approaches have been used. Some programs, such as DRAW and CRAWL, are essentially file-in, file-out, although with some possibilities for the user to initialize variables, etc., at the beginning of the session. Others, particularly POST and IFORM, rely heavily on LISP's workspace structure.

As a consequence, two different methods for maintaining structured data are both used:

>a block of data (for example, one or a few "relations," or assignments of a number of "properties" to a number of "objects") may be stored as text files between sessions, and loaded into the data base when needed during a session. If data are changed, a new text file has to be produced, but this need only happen at the end of the session, or occasionally during the session but then only for reasons of backup and reliability. This method will be called *residential* storage of the data base;
>alternatively, data elements (such as individual records in a relation, or property assignments to one "object") may be stored primarily as a segmented disk file even during the interaction session. Each data element is read into the workspace when it is needed, and if changed, the change is immediately performed on the disk file. This method will be called *external* storage of the data base.

Residential storage is the classical modus operandi in a LISP environment, and is very strongly supported by the INTERLISP system itself, which therefore is to be viewed, among other things, as a database system in the present context. External storage is sometimes advantageous, particularly when relatively little processing is performed on each data element, and when data and their updates are to be seen simultaneously by several users.

Other useful properties of the INTERLISP system

Some other properties of the INTERLISP programming system which were significant for the development of this system, are:

>the very advanced support for program development activities: administration of programs, debugging, etc.;
>the possibility to store parameter structures in the built-in data base (within the LISP workspace) and obtain services for the maintenance of this data base;
>systems-programming facilities, such as easy interface to assembler code and to operating-system calls.

The major negative property of the system has been the relatively long time required to learn it. Since the language and the programming system is intended as a tool for the professional programmer, its high power must be paid by a relatively long learning time.

Performance

Since the intended purpose of the present project has been to develop an experimental system, which could be easily modified, but which also could be used within our group, the question of how much emphasis we should place on performance has recurred in the course of the project. Better performance can be achieved at the cost of more work and (often) a less transparent program. In particular, the use of LISP for major parts of the system represents a very high priority for ease of development and maintenance, perhaps with a danger of slow performance.

Have we then obtained performance problems as a result of this strategy? This depends on how you look at it. Like most time-sharing systems in research environments, our computer system is sometimes badly overloaded, and the continued development work on parts of LOIS is not the least of reasons. However, if one judges the response times and general behavior of the LOIS system as seen by a user at times when the system is reasonably loaded (i.e., not thrashing), it seems that all major parts of the system are sufficiently quick for their intended purpose. The parts where response times are critical are the ones which have been programmed in assembler, and they form a relatively small part of the total software. The other parts, which have been writte LISP, are characterized either by a small amount of processing (although often of considerable complexity), especially in IFORM, or by a semi-batch mode of usage where longer execution times are tolerated especially if advantages of flexibility are offered instead.

This point may be illustrated with some figures. The CRAWL text formatter, entirely written in LISP, is about ten times as slow as the RUNOFF system, written in assembler. One should then remember that:

>CRAWL provides certain additional services, such as variable-width fonts;
>no attempt has been made to optimize CRAWL. A preliminary survey of what can be done indicates that there are several simple things one can do in the innermost loops, using short assembler routines;
>the timings were made using the regular INTERLISP compiler; the block compiler could be used to speed it up.

In some cases, the first version of a program turned out to be too slow and had to be rewritten to gain speed. This only happened for a few, small programs (such as the low-level mail receiving program) and may to a large extent have been due to the programmer's short experience of LISP programming.

Continued strategy

In summary, we believe that the chosen implementation strategy has been a good one. Our continued strategy will be to develop additional facilities in LISP, and gradually improve the efficiency of existing facilities by a number of measures:

>optimizing within the LISP context;
>transfer by semi-automatic means to another programming language (for programs which do not need all of LISP's facilities);
>transfer to smaller and cheaper processors for dedicated purposes, where CPU requirements may become less of an issue.

ACKNOWLEDGMENTS

Many members of our group have helped with good ideas and constructive critique, in particular Jim Goodwin, Erland Jungert, John Walters, and Jerker Wilander, and Peter Fritzson and Dan Strömberg who also participated in the programming.

The variable-font printout system relies on several kinds of hardware built at the Electrical Engineering department of our university and in the Lysator society: the LYS-16 computer, the T2 special-purpose graphic terminal used for editing fonts, and others. In particular, we are grateful to Olov Fahlander for building the T2 and to Robert Forchheimer for a continuous interchange of ideas and information.

The project owes gratitude for the body of ideas and the software that we have inherited from the MIT Artificial Intelligence Laboratory, from Xerox Palo Alto Research Center, and from Bolt, Beranek and Newman, Inc. in Cambridge, Mass.

REFERENCES

1. Morgan, H. L. , *Office Automation Project*. Proceedings of the 1976 NCC Conference.
2. Kernighan, B. W. et al., *Unix Time-Sharing System: Document Preparation*. The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.
3. Sandewall, E., *A Description Language and Pilot-System Executive for Information-Transport Systems*. Proceedings of the Fifth International Conference on Very Large Data Bases, Rio de Janeiro, 1979.
4. Hammer, M [illegible], *A Very High Level Programming Language for Data Processing [illegible]ations*. Comm. of the ACM, Vol. 20, No. 11, Nov. 1977.