Reprinted from Proceedings of the First European Conference on the Practical Applications of Lisp. Cambridge, UK, pages 183-191.

RFT Design System - Experiences in the Development and Deployment of a Lisp Application

Jukka K. Nurminen
nurminen@rc.nokia.fi
Department of Knowledge Technology
Nokia Research Center
P.O. Box 156, SF-02101 Espoo, Finland

Abstract

In this paper we discuss the use of Lisp in an intelligent design application. The RFT system has been developed to support the design of analog electronic devices. It facilitates the use of complicated numerical design tools, such as simulators, and provides a framework which can be completed with knowledge modules to assist the user in certain design tasks.

In addition to a brief presentation of the RFT system, we discuss some practical issues of Lisp software development including portability considerations, graphical user interfacing and object-oriented programming. We also present some comments from the users of the RFT system to estimate the suitability of a Lisp application for real use.

1 Introduction

Lisp is often used as a prototyping language and until recently few Lisp programs have been in real use. Lack of standardization and extensive hardware requirements have been some of the main reasons that have prohibited a wider use of Lisp. Now this situation has changed. Standardization, dropping hardware prices, and advances in compiler technology make it possible to seriously consider Lisp as an implementation language of software systems.

In this paper we discuss some of our experiences of using Lisp to develop software for industrial use. The discussion is concentrated mainly on issues that we have found critical or problematic in a project that has developed a knowledge-based design framework called RFT [6, 8]. Since RFT is in use in several design groups inside the Nokia Corporation we also discuss some points that the users have considered important.

The rest of the paper is divided into three major sections. Section 2 briefly presents the basic ideas and the operation of the RFT system. Section 3 concentrates on practical experiences in the development of the system. We discuss some of the reasons for using Lisp in the implementation of the system, point out some of the benefits and problems of this decision, and present some of our solutions. Finally, in section 4 we summarize the discussion and present some conclusions.

2 RFT design system

2.1 Motivation

Analog electronic design requires extensive use of numerical tools. Simulators and other analysis and design tools are heavily used during the design cycle. However, these tools are often difficult to use.

The user has to define the problem in a special way for each tool, which requires a lot of routine work as well as expertise. Moreover, the conceptual level of the interfaces is often wrong. The design engineer can most efficiently work with familiar concepts, such as design diagrams. However, when he analyzes the design he has to formulate the same problem with other concepts for the design tool. For instance, when using a block level simulator he has to express his ideas using simulator blocks. Some other tool may require him to formulate the problem using mathematical equations. This kind of conversions cause extra burden to the designer and distract his concentration from the actual design problem.

Difficulties in tool usage also make it hard to use a number of tools in a co-operative way. Since a lot of details has to be taken care of before a tool can be started, it is difficult for a tool to state a problem and invoke some other tool to solve it.

2.2 Intelligent use of design tools

One way to help with the above problems, is to include into the design environment expertise that takes care of details of tool usage. This is one of the basic ideas of RFT. Expert users define knowledge of the design process and of the design tools, and enter this knowledge into the system.

RFT uses this knowledge in model and problem formulation and in conversions between different representations. Combined with a graphical user-interface this allows the user to work with familiar concepts. The mathematical details of the design tools and the necessary conversions are hidden from the user allowing him or her to concentrate on important design decisions.

The necessary conversions are not simple. For instance, block-level simulators usually operate with ideal components. Since the real components have many non-ideal features, they have to be modeled using a number of ideal simulator blocks. As an example, an amplifier can be modeled using an ideal amplifier block and a white noise random generator. Figure 1 shows a simple functional level design diagram and some simulation models that can be used to model it. Which of the models is selected, depends both on the topology of the design diagram and on the parameter values of the functional blocks. This kind of conversion knowledge is used when a suitable simulation model is generated from the user defined design diagram. For some components the conversions depend also on the goals of the simulation. This is used to make the simulations more efficient by removing irrelevant blocks. Thus the conversion not only facilitates the use of the tools but also improves their performance.

2.3 Higher level operations

Besides increasing the user-friendliness the above features facilitate the development of another layer of automation on top of the basic tools. The basic tools give the user feedback of the effect of design decisions but they do not directly help in finding good design solutions. Operations for giving design suggestions and for searching for good designs are needed to increase the automation level of the system. The implementation of this kind of features is fairly easy since the user does not have to worry about the details of the actual analysis. Automatic conversions and knowledge of the use of the design tools take care of these.

A major class of high level operations are related to design synthesis. The operations include both optimization and rule-based design. One of the main goals in the implementation of these techniques has been to create general tools that can be used for a number of different tasks in the design process.

Conventional optimization algorithms are usually effective only in some specific problem types and most of the algorithms require a mathematically rigorous problem formulation. To avoid these limitations we have used the simulated annealing algorithm [9], to perform the optimization tasks. It is more flexible and robust than conventional optimization methods mainly because it is not dependent on the form of the objective or constraint functions. Its drawback is that the optimality of the results can only be guaranteed probabilistically. For reliable results, it requires a large number of iteration steps and, respectively, a large number of function evaluations. This makes it impossible to use the algorithm when the computations are slow, e.g. with simulations. However, in RFT many analysis are performed with problem-specific, fast routines which allows the successful use of the algorithm.

Although the above optimization concept is widely applicable, it cannot be used for problems which defy mathematical formulation or which require an impractically long solution time. Design experience and heuristics are therefore needed to supplement the optimization features. For smooth operation it is important that the rulebases representing this knowledge are closely integrated with other parts of the system, especially with the user interface and with the analysis tools. To be able to do this, we transform the if-then rules to Lisp functions. A special interaction language is used to define how these rule-functions behave when they need access to values that cannot be deduced in the rulebases. For instance, missing values can be asked from the user, derived by some analysis or read from the design diagram.

2.4 RFT-design system

Figure 2 shows the main parts of RFT. Because the application area of RFT is radio frequency design, the analysis tools are intended for this field. However, by changing the toolsets and design knowledge it would be possible to apply RFT to the support of other design tasks as well.

The basic analysis tools, Aplac simulator [4], TOPSIM simulator [11], cascade analysis and transfer function analysis tools are all connected to the object-oriented design representation through a layer of tool knowledge. The purpose of the tool knowledge is to take care of the details of tool usage.

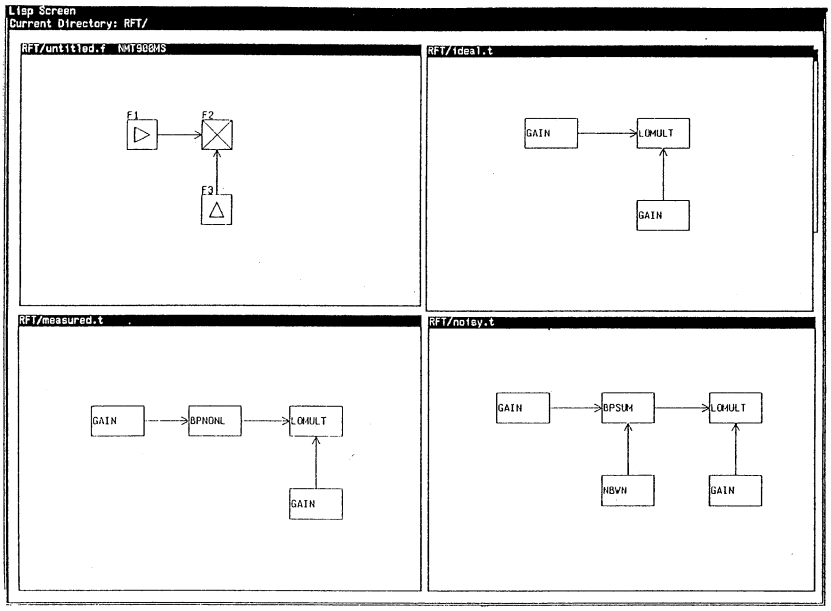

Figure 1: A screendump showing a functional level design diagram with some possible simulation models

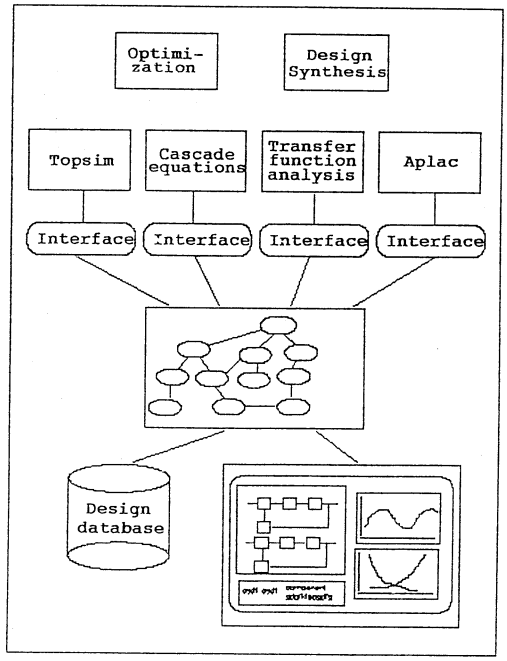

Figure 2: The architecture of RFT

On top of the basic analysis tools are higher level operations, rulebased design synthesis and optimization. These operate in co-operation with one or several basic tools. They may also refer to the design diagrams that are being studied.

The graphical user interface allows users to edit design diagrams and activate different tools. It also takes care of the visualization of the results. Figure 3 shows a typical example of the use of RFT.

2.5 Current status of RFT

Currently RFT is in use in several product development groups in Nokia Corporation. The computer assisted analysis tools are the most frequently used components of the system.

Design optimization is currently used actively only in the selection of system level design parameters. In this task it has given results that are comparable to the designs developed by experienced engineers.

The rulebased design synthesis part is so far fairly simple. More design expertise needs to be stored into the system to make this feature really useful. We are currently developing a design knowledge acquisition tool to help in this task.

3 Development and deployment issues

In this section we will discuss some issues that have been significant in the development of RFT. With some 40,000 lines of Lisp code, RFT is a fairly large system. Graphical interaction with the user and cooperation with external programs are important parts of the system. In the following discussion we will especially emphasize these issues.

RFT has been implemented with Lucid Common Lisp. During the project different Lisp versions have been used; initially version 2.0, current release is running both on version 2.1 and on version 3.0, and the next major release of RFT is intended to be running on version 4.0. The different platforms in use are Sun, Apollo, and HP workstations.

3.1 Some language selection criteria

3.1.1 Fast prototyping

At the beginning of the project, the goals and objectives of the RFT system were not very clear. The would-be users had a fairly vague idea of the possibilities of computers and of knowledge technology. The developers, on the other hand, were not very familiar with the needs of design engineers. In order to be able to better match the ideas of the different parties, prototyping was considered necessary to provide a concrete basis for further development.

Lisp is often claimed to be a good language for fast prototyping [2]. The language constructs and efficient programming environments support fast program development. This makes prototyping more feasible by allowing large amounts of discarded code to be rewritten quickly.

Another feature which further supports prototyping is a library of reusable Lisp routines that has been developed in our group. This facilitates especially the user interface development. The library allows different projects to share code and thus reduces the often quite time-consuming user-interface development effort.

3.1.2 Symbol manipulation

RFT was intended to provide a framework which the users can tailor to their own needs. This is done by defining knowledge both of the design process and of the design tools. To be able to efficiently process these definitions flexible symbol manipulation features are needed.

Another important use of symbol manipulation is transformations between different representations used in the integrated design tools. For instance, converting a mathematical formula to a representation required by FORTRAN compiler.

3.1.3 Personal preferences and experience

The attitudes of the persons involved were in favor of Lisp. All persons had earlier experience on using Lisp either in a Lisp machine environment or on a personal computer. In spite of the different platforms, all preferred working with Lisp compared to other language alternatives such as C.

3.1.4 Portability

An important issue in developing software for industrial use is that different users want to use the application in different machines. Its difficult to justify an investment to a new computer only to be able to run some new software. Therefore an important practical requirement is that the software should be portable.

Although Common Lisp has been standardized, important language components, such as user interfaces and object-oriented extensions, are not standard-

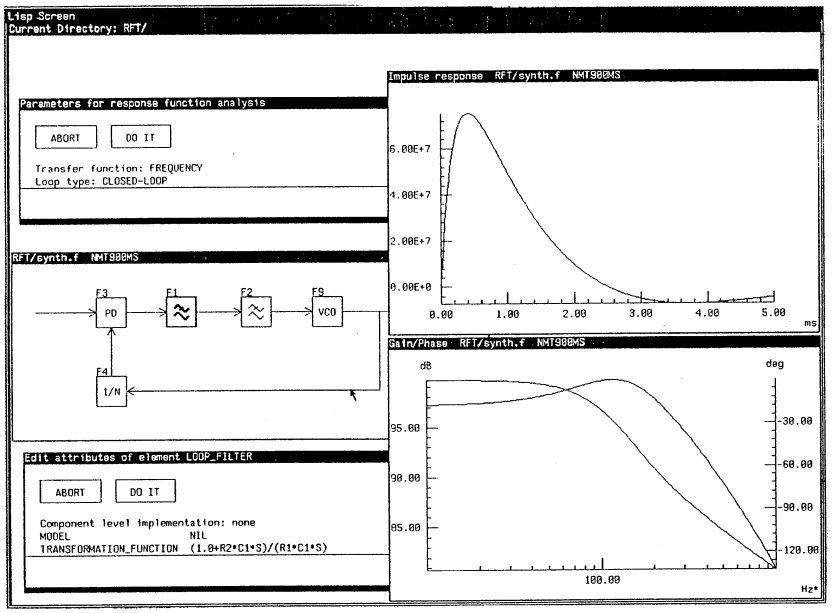

Figure 3: RFT screen during a typical working session

ized or the standards are so recent that few Lisp implementations support them yet. For this reason we have had to pay special attention to the portability of our software. Techniques for achieving this are discussed in greater detail in the following sections.

3.2 Object-oriented programming

The RFT system is based on the object-oriented paradigm. The main reasons for this are that

- the design objects of hardware design (e.g. modules, components, and connections) can be naturally modeled as objects in the programming language
- object-oriented programming supports the development and maintenance of large software systems.

When the RFT development was started a major problem was that the object systems were not standardized. This is going to be solved with the emerging Common Lisp Object System (CLOS) standard [5]. Since the standard was not available when the development of RFT started, an in-house object system called NOS (Nokia Object System) [7] was used.

NOS was based on several key ideas:

- It was developed in standardized Common Lisp, which makes it completely portable.
- The syntax of NOS was selected to be as near as possible to the CLOS standard proposal available at that time.
- NOS was intended to be very simple containing only the most important features of object systems, such as generic function calls and simple inheritance.

Although the implementation of NOS is fairly simple, it has worked reasonably well. The initial idea was to use NOS as an intermediate step before moving to CLOS. However, because of the positive experiences with NOS, this has to be reconsidered when CLOS implementations become available.

The main problem of NOS is that it does not support dynamic inheritance. Therefore modifications to the class hierarchy are tedious. The addition of a new slot or method to a class requires that all its subclasses have to be re-evaluated or re-compiled. In a large class hierarchy this can take several hours of time. This problem does not prohibit the use of NOS. However, it makes it difficult to follow good object-oriented

programming practice. Programmers try to avoid the extensive re-compilations by minimizing modifications in any super classes. This often results in solutions that are both clumsy and unnatural.

Another problem, which is of historical origin, is that the number of separate classes is too small. Because the class definitions required a lot of memory space, the object system in HP that was used in the early phases of RFT development could handle only a very limited number of classes. To reduce the number of separate classes we used parameterized abstract classes to represent sets of different object types. Although this problem has been solved in NOS, some of the older parts of RFT are not very clearly structured because they are written in a partly object-oriented and partly functional programming style.

The usual problem with object-oriented systems is disk storage of objects. In RFT the relevant data of the object instances is written into normal operating system files in an application specific way. This solution is not very good for maintenance since the addition of a new slot to a class requires also modifications to the storage routines. Additionally, this storage system is not very suitable to data transfer between different applications. Our earlier solution to store the objects in a relational database (Oracle) was better in this respect, but it turned out to be too slow to be acceptable to the users.

3.3 User interface

Graphical user interface is an essential component of an interactive design environment. Its importance is even emphasized in hardware design because many design engineers are used to working with commercial schema editors which provide sophisticated user-interface services.

In the development of RFT the selection of user interface software was similar to the selection of the object system. There was an emerging standard, X Windows, but when the project was started no completely standardized version of X Windows toolkits existed, and a commercially available Lisp interface to X Windows was missing.

Therefore we abandoned X Windows and implemented the user interface using the Windows Tool Kit of Lucid Lisp. The benefits of this solution are that

- the Windows Tool Kit is available in all machines that are supported by Lucid. This is an important advantage since the same RFT source code can be used in Apollo, HP, and Sun workstations.
- the Windows Tool Kit is intended for Lisp use and therefore provides a natural interface to Lisp code.

Some of the most obvious disadvantages of our solution are

- Lucid Windows Tool Kit uses a machine dependent window system for the actual graphic operations. In Apollo and Sun this concept works well, but in HP where Lucid Windows Tool Kit has been build on top of X Windows the speed efficiency is bad. This is especially problematic in tasks where constant visual feedback to the user is needed, such as in positioning a component. The likely reason for this problem is that two large processes, Lisp and X Windows, are competing for the available memory space which increases the virtual memory load.
- X Windows provides a much larger variety of services than Lucid Windows Tool Kit. However, these more sophisticated features are seldom needed.

To further support the user interface development we have developed on top of the Windows Tool Kit an object-oriented, higher level window system called AIGT [1] (Application Interface Generation Tools). In addition to simple window classes, e.g. for messages, menus, and forms, AIGT consists of several classes that can perform fairly sophisticated operations. For instance, a block-editor can be used to draw block-diagrams, a grapher to visualize and modify graph and tree structures, and a coordinate-window to draw curves. This kind of general classes that can be inherited to the application classes have increased code reusability in different application projects to a great extent.

3.4 Some development problems

3.4.1 Lisp versions and portability

In general, the above concept of using Lucid Lisp, Lucid Windows Tool Kit and NOS object system has worked well. In theory all software would be completely portable between the different platforms and the interface should look exactly the same in each installation.

In practice, however, there are small differences (e.g. in external function interface and in pathname handling) between different machines running the same Lisp version. The differences are very small

so that there is only about 20 - 30 lines of code (of the overall 40,000) that has to be changed. However, although the number of differences is very small, noticing and correcting them has sometimes required a lot of work.

Most of the differences between successive Lisp versions are documented and are thus easy to find. Modifications in, e.g. process handling between versions 2 and 3 of Lucid Lisp, have forced us to modify some parts of our code. Modifications are made more complicated by the fact that different vendors have very different distribution schedules. This means that we have to keep the software compatible not only with the different machines but with the different Lisp versions as well.

3.4.2 Customer service

Like all software new Lisp versions have bugs and unexpected features. Good customer service is important to assist with this kind of problems.

Different vendors have different approaches to customer service. Sun has a centralized facility for bug reports, an electronic mailbox where problems can be directly sent. This kind of a system seems to work well.

HP and Apollo have a hierarchical support organization. Problems are first reported to the national customer support personnel. If they cannot solve the problems, they forward the problems to the next level in the organization hierarchy and so on, until someone comes up with an answer. In this way getting an answer can take a long time. Another problem with this kind of organization is that it can cause misunderstandings because the original problem changes its form when it is forwarded up in the hierarchy.

3.4.3 Program development support tools

The Lisp development environment provides nice features for program development. For instance, the incremental compiler and the integrated debugger increase the programming productivity considerably. However, problems arise when the services provided by the Lisp environment are not adequate. Since Lisp, in spite of the operating system and external function interfaces, is a fairly closed system, the use of Unix tools for program development support is not easy.

For example, the use of version management tools, such as RCS [10], requires modifications both to the way the editor is used to find files and to the way files are loaded and compiled. A fairly large interface has to be written to be able to take advantage of all the features of RCS. The same applies to the use of other Unix-software such as relational databases.

3.5 User acceptance

3.5.1 Cost

Users attitudes have been fairly positive towards the use of Lisp. The fairly small additional cost of the runtime Lisp environment has not been a critical factor so far.

The main cost factor is the hardware that is needed to use the application. However, in our electronics design application this is not a very big problem. Most design groups already have workstations that are needed to run various CAD-programs. To speed up the simulations, most of the workstations are equipped with larger amount of memory than the standard configurations. Therefore, all of the design groups that are currently using RFT were able to start its use without extra hardware investments.

An additional feature that affects the hardware cost is, that Lisp requires so much resources that it is slow for more than one person to use a workstation at the same time. However, because many CAD-tools for electronics engineering require some graphical interaction with the user via the system console, engineers are used to this.

3.5.2 Efficiency

Because of the interactive nature of the application, speed is one of the most important features for user satisfaction. The most important factor affecting the speed is the amount of memory. The larger the memory space the less time is spent on swapping data to the auxiliary memory. Interestingly, memory requirements in different machines with the same processor are not the same; in some machines the performance is acceptably fast with only 6 MBytes of memory but in other ones 7 MBytes is still not enough for smooth performance.

Garbage collection, often claimed to be one of the major problems of Lisp, has not been a real problem. In a design application the user spends most of the session time thinking and considering different design alternatives. Therefore breaks or short time performance drops because of the garbage collection are not harmful. The main problem is that the machine should signal the user clearly when garbage collection is taking place. If this is not done clearly enough,

users try to click the mouse and press different buttons in order to make the machine respond. When garbage collection is completed and the normal operation continues this kind of buffered input can cause unexpected operations and accidental damage.

3.5.3 User extensions

Normal use of RFT is completely menu-driven and the users do not have to know anything about Lisp to be able to use the system. However, if users want to change the behavior of the system and tailor it to different tasks, they have to modify the design knowledge.

Design knowledge definitions are extensions of Lisp which have been implemented using macros. Although the structure of the definitions is fairly simple, an elementary knowledge of Lisp would be useful for those persons who modify them. Problems are especially likely to arise when the user makes an error in a definition, and he or she should understand what is wrong and how it should be fixed.

A minor problem related to this kind of modification is that the new or modified knowledge defined in this way can only be used in the evaluated form. Compilation is not possible since the run-time Lisp environment does not include a compiler.

4 Conclusions

In general, Lisp has turned out to be a good language selection for our application. It has fulfilled the main objectives fairly well. The reactions of the RFT users have shown that Lisp applications can be successfully delivered and used in real design operations.

Most of the problems that we have encountered can be attributed to two factors: missing standardization of critical parts of Lisp and the still fairly small size of the user community. The recent CLOS standard and the eventual user interface standardization are likely to solve most of the portability problems. Standardization hopefully guarantees that these features are implemented in an efficient way and integrated smoothly into Lisp environments. The increasing number of Lisp users is likely to widen the experience of the support personnel and stimulate the development of tools to support Lisp program development.

References

[1] *AIGT Application Interface Generation Tools,* Users guide, Nokia Research Center, Espoo, Finland, July 1989.

[2] Fahlman, S. F., "Common Lisp," in Traub, J. F., Grosz, B., Lampson, B. W., and Nilsson, N. J. (eds.), *Annual Review of Computer Science*, Vol 2., Annual Reviews Inc., Palo Alto, California, 1987.

[3] Gross, J., "Two vendors tout CLIM standard," *Computer Resekker News*, July 24, 1989.

[4] Heikkila, P., Valtonen, M., and Pohjonen, H., "Automated Dimensioning of MOS Transistors without Using Topology-Specific Explicit Formulas," *Proceedings of the 1989 ISCAS*, Portland, Oregon, May 1989.

[5] Keene, S. E., *Object-Oriented Programming in Common Lisp, A Programmer's Guide to CLOS,* Addison-Wesley Publishing Company, USA, 1989.

[6] Ketonen, T., Lounamaa, P., and Nurminen, J. K., "An Electronic Design CAD System Combining Knowledge-Based Techniques with Simulation and Optimization," in *Proceedings of the Third International Conference on Applications of Artificial Intelligence in Engineering*, Palo Alto, California, August 9-11, 1988.

[7] Maamies, T., "Implementation and Evaluation of an Object-Oriented Programming System and an Object-Oriented User-Interface Toolkit (in Finnish)," Licentiate's thesis, Laboratory of Computer Science, Faculty of Information Technology, Helsinki University of Technology, 1989.

[8] Nurminen, J. K., "Coupling Symbolic and Numeric Computing in Knowledge-Based Systems: An Application to Electronics Design," Licentiate's thesis, Systems Analysis Laboratory, Faculty of Information Technology, Helsinki University of Technology, 1989.

[9] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., *Numerical Recipes - The Art of Scientific Computing*, Cambridge University Press, 1986.

[10] Tichy, W. F., "RCS - A System for Version Control," *Software - practice and experience*, Vol. 15(7), July 1985.

[11] *TOPSIM III*, Computer Software, Torino Polytechnico, Italy, 1983.