UC Irvine
ICS Technical Reports

Title
Rapid software prototyping

Permalink
https://escholarship.org/uc/item/217238b5

Author
Smith, David Andrew

Publication Date
1982-05-12

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no. 187

Rapid Software Prototyping

by

David Andrew Smith

TR# 187

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Information and Computer Science

Computer Science Department
University of California
Irvine, California 92717

Technical Report Number 187
May 12, 1982

This work was supported in part by the Defense Advanced Research Projects Agency of the United States Department of Defense under contract MDA-903-82-C-0039 to the Irvine Programming Environment Project. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

(C) Copyright 1982, David Andrew Smith, All Rights Reserved

DEDICATION

To John David Smith and Minnie Louise Elizabeth Smith,

my father and mother.

CONTENTS

ACKNOWLEDGMENTS v
ABSTRACT. vii
CHAPTER 1: INTRODUCTION. 1
DEFINITION AND MOTIVATION. 1
AN EXAMPLE 2
BENEFITS OF RAPID PROTOTYPING. 3
SELECTION OF AN APPROACH 5
SCOPE OF THIS THESIS 8
CHAPTER 2: PRINCIPLES OF RAPID PROTOTYPING 14
MOTIVATIONS FOR RAPID PROTOTYPING. 17
The Software Lifecycle. 17
Improved Feedback to Requirements Analysis. . . . 19
Validation of Novel Designs 20
One-Time Applications 22
Rapid Prototyping During Maintenance. 22
WHAT TO SACRIFICE FOR RAPID PROTOTYPING. 23
Efficiency. 23
Scope of Problem Size 24
Functional Capability 24
CHAPTER 3: METHODS OF RAPID PROTOTYPING. 26
EXAMPLES OF RAPID PROTOTYPING. 27
Automated Flight Service Station. 27
Custom Microprogram Assembler 32
St. Lawrence Seaway Traffic Control System. . . . 35
METHODS OF RAPID PROTOTYPING 36
Reduction of Scope. 36
VHLL's and Program Generators 38
Reusable Software 38
Simulation. 39
Reconfigurable Test Environments. 40
RAPID PROTOTYPE PROGRAMMING. 41
CHAPTER 4: RAPID PROTOTYPE PROGRAMMING ENVIRONMENT . . 42
THE TARGET LANGUAGE, ADA 43
PROTOTYPE DEVELOPMENT IN CASTOR. 46
SWITCHING CONTEXTS DURING PROBLEM SOLVING. 47
Calling Forms in Castor 47
Aide Management of Calling Forms. 49
PROGRAMMING BY REFINEMENT. 51
Program Attachments 56
AN EXAMPLE: THE EIGHT QUEENS PROBLEM. 64
CHAPTER 5: THE USE OF CALLING FORMS. 69
CALLING FORMS. 69

Lexical Considerations. 71
Subprogram Calling Forms. 74
Language Extension. 75
Programming Worlds. 77
Program Transformations 78
THE USE OF PARAMETERS. 79
SELF-REPLACING FORMS 81
CALLING FORM MACRO EXPANSIONS. 87
Macro Definitions 88
Data Types Used In Macros 89
Syntactic Validation Of Macro Expansions. 91
Generating Program Fragments. 92
Substituting Into Program Skeletons 93
Aggregates of Nodes 95
Additional Features 98
PROGRAMMING WORLDS 101
CHAPTER 6: INTERACTIVE PROGRAM MANIPULATIONS 108
EXPLOITING DATA TYPES. 109
Reducing Verbosity of Data Types. 110
Type Checking 112
Uniform Notation. 113
INTERACTIVE PROGRAMMING CONSIDERATIONS 118
Automatic Command Completion. 118
Deeply Nested Calling Forms 119
CHAPTER 7: RELATED WORK. 120
PROGRAM DESIGN LANGUAGES 120
ANNOTATION OF PROGRAMS 122
INTERACTIVE DESIGN SYSTEMS 123
TRANSFORMATION SYSTEMS 126
VERY HIGH LEVEL LANGUAGES. 128
CHAPTER 8: CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK. 130
RAPID PROTOTYPING. 130
CASTOR AS A PROGRAM DESIGN LANGUAGE. 132
THE ADA PROGRAMMING LANGUAGE 135
SPECIFICATIONS 137
BIBLIOGRAPHY. 138
APPENDIX A: AN EXAMPLE OF RAPID PROTOTYPING. 144
OBJECTIVES OF THE PROTOTYPE. 145
A SET OF FUNCTIONS TO SELECT FROM. 147
THE MODEL SYSTEM 150
Basic Concepts. 151
File Management Commands. 152
Document Templates. 154
Editor Commands 154
Alarm Functions 155
Document Forms In The Model System. 156
FUNCTIONS SELECTED FOR THE PROTOTYPE 160
THE TOP LEVEL OF THE PROTOTYPE 161
A REFINEMENT OF THE PROTOTYPE. 164
MODULARIZATION OF THE PROTOTYPE. 170
USE OF MACROS IN THE PROTOTYPE 173

ACKNOWLEDGMENTS

I would like to thank Tim Standish, my advisor, for his guidance, technical advice, and meticulous reading, and particularly for his continuous patience and enthusiasm which made completion of this work possible.

I would also like to thank the members of my committee, Rick Granger and Dennis Kibler, for their reading and helpful suggestions.

I would like to acknowledge the implementors of the first UCI experimental interactive Ada system. Without this system built by Preston Bannister, Dan Eilers, John Long, Tami Taylor, Stephen Willson, Kathy Velick, and Steve Whitehill, the implementation of Castor would have been physically impossible. A full-screen editor written in Ada by Scott Ogata was used as the basis for the editor component of the prototype "electronic office" described in the appendix.

Jim Meehan has made available a powerful set of tools in the UCI LISP/MLISP system. He has provided me with consistent, personal help with the system on many occasions.

This work was accomplished while I was a Member of the Technical Staff and a Staff Engineer of the Hughes Aircraft Company, Ground Systems Group, and a holder of a Fellowhsip under the Hughes Staff Doctoral Fellowship Program. I also received support on Professor Standish's National Science Foundation Grant MCS75-13875 entitled "Interactive Program Manipulation." Computer time for the development of Castor was provided by the Irvine Programming Environment Project, under DARPA contract MDA903-82-C-0039.

I am grateful to Debra, my wife, and to Ronalyn Choco for their assistance in proofreading the manuscript.

I feel it is proper to acknowledge the hand of the Lord who has blessed me temporally and with needed inspiration during this time.

ABSTRACT

Rapid Prototyping is an approach to software development which emphasizes quick implementation of a working program. This dissertation makes two principal contributions. First, it provides concepts, techniques, and a philosophy of Rapid Software Prototyping and characterizes the benefits and limitations of its use. Second, it makes a contribution to programming environments which support Rapid Prototyping. An experimental language, Castor, is described which was implemented to validate this approach in the prototyping of Ada programs*. The following summarize the main results of this research:

1. A statement of the purpose and value of Rapid Prototyping: Rapid Prototyping provides accelerated feedback to the early stages of analysis in the software lifecycle. This can be of great benefit when there are areas of risk that only experience with a working system can resolve.

2. A statement of the limitations of Rapid Prototyping: Rapid Prototyping cannot show the behavior of the final system in all respects. Careful planning is therefore necessary to determine the objective of the prototype and what sacrifices can be made in areas of low risk.

3. Techniques for Rapid Prototype Programming: Castor is both a Program Design Language (PDL) and an implementation language. The PDL nature of Castor arises from the use of free form descriptions called "calling forms." An agenda of undefined calling forms is provided interactively. Contributions in this area are that:

   a) Castor implements a refinement paradigm for the new language, Ada;

---

*) Ada is a trademark of the United States Department of Defense.

b) Castor macro facilities are easy to learn and remember; and

c) the Castor macro language is independent of the underlying program representation.

4. A stock of ideas for an "Ada laundry": An Ada laundry allows the user to relax temporarily the rules of pure Ada. This helps compensate for aspects of Ada which orient it more toward long program life than short term ease of expression.

Castor was used to build a prototype of moderate size which is described in an appendix.

CHAPTER 1

INTRODUCTION

DEFINITION AND MOTIVATION

Rapid Software Prototyping is an approach to software development which emphasizes quick implementation of a working program. Software Prototyping is a valuable tool for the same reasons that prototyping is important in other fields of engineering -- a prototype gives the system users and implementors experience with a working version of the proposed system at an early point in the development cycle. This early feedback makes it possible to adjust, if necessary, the system concepts and goals before large investments have been made in a production-quality implementation. As a software development tool, Rapid Prototyping can reduce wasted implementation effort and can help make the final product more responsive to the user's needs.

AN EXAMPLE

To illustrate this process, consider how one might develop software for a desk-top computer to automate the data handling normally done on paper in a business office. One is given a statement of functional requirements stating, for example, that the system must manage meeting agendas, appointment calendars, telephone number lists, and so on. A specification would then be developed, possibly in conjunction with the users, defining basic data structures, operations, and patterns of interaction between the computer and its operator. Supposing this to be a new application, we may perceive a certain amount of risk in whether or not the requirements statement has correctly captured the true needs of the prospective users and whether the system we have specified will meet those needs.

We can use prototyping to respond to these perceived risks by implementing a working system or subset of the system as soon as possible to get user feedback. We could do this using a high-level, interactive language on a host timesharing system separate from the required target machine. This host system could be exploited in a number of ways -- using, for example, its virtual memory, its file system, its inter-user communication features, and other of its utilities, as required. By cutting corners and perhaps by implementing a subset of the system functions --

appointment calendars and telephone number lists, for example -- we could expose users to behaviors of the intended system at a very early point in the development.

The users could then evaluate the prototype by using it on real or typical problems. They would be able to assess the effectiveness of the user interface and the functions provided and possibly they would discover unanticipated problems with the system in actual use. The users would also be in a good position to evaluate whether the system requirements had been adequately captured and correctly articulated. Using a prototype, this valuable experience is gained before embarking on further and usually costly steps of design, review, and documentation -- practices required for a product with significant implementation constraints and a long expected maintenance lifetime. And the risk has been reduced that the user might not like the final system or might find that it doesn't serve his needs adequately.

BENEFITS OF RAPID PROTOTYPING

We view Rapid Prototyping as a means of reducing risk in a software development. There is, of course, no way to eliminate all uncertainties before actual completion of a system. However, in a novel system there may be certain areas of particular uncertainty or risk. We may have new algorithms or hardware devices, new modes of user

interaction, or the introduction of computer usage into a previously unautomated environment. We may be uncertain whether system requirements have been articulated correctly or whether a system built to given specifications will actually meet the user's needs. What is more, actual introduction of the system into its new environment may significantly change the environment in unforseen ways.

In summary, there are times when it may pay to observe and experiment with the behavior of the system. Careful formulation of requirements, specifications, and design are important. But behavioral feedback may reveal information that is difficult to discover by analysis of a static system description. This is particularly so when the user is not trained to understand technical system descriptions. In the traditional lifecycle model -- which in many cases is firmly embodied in procurement policy -- this behavioral feedback becomes available only at the end of a lengthy development. Rapid Prototyping is a way of shortening this feedback path in key risk areas before large investments have been made in the development.

The experience gained in building a system prototype can also be applied profitably during full-scale implementation. The prototype can serve as a vehicle for experimentation and learning if the same implementors are involved in the follow-on work. Prototyping techniques may

also prove useful for exploring alternative problem solutions or providing quick solutions to one-time problems.

We note that there is growing recognition of Rapid Prototyping as an important research area in computer science. For example, as of this writing a Workshop on Rapid Prototyping has been held by ACM SIGSOFT and the National Bureau of Standards [Workshop 82a, Workshop 82b]. Rapid Prototyping has also been mentioned as a potential thrust area for the Software Technology Initiative being examined for possible sponsorship by the Deputy Undersecretary of Defense.

SELECTION OF AN APPROACH

In order to restrict the problem area which our research addresses we have chosen to investigate Rapid Prototyping systems which have the following characteristics. A Rapid Prototyping system must:

1. provide the earliest possible behavioral feedback by creating executable system descriptions early in the software lifecycle;

2. provide the means to achieve a smooth, practical transition from initial system descriptions into executable Ada programs;

3. minimize the difficulty of learning how to use the system; and

4. provide a rapid, powerful programming capability with the ability to trade off ease of expression for program performance.

A variety of available approaches satisfy these characteristics and each approach has particular research issues and background literature. Among these approaches are:

Software Reuse. This includes the technical issues of describing and cataloging software components. Also of concern are mechanisms for retrieving and composing components and integrating them to solve a given problem.

Parameterized System Generation. When a class of programs is well understood, it is sometimes possible to view them as specialized instances of a common abstract model. Such programs can be automatically generated, given a set of appropriate parameter values.

Programming By Refinement. This general purpose approach is to provide a tool set for supporting

program development by stepwise refinement. This can be done to aid fast design and implementation or instead can emphasize the structuring and recording of decisions for later use during program maintenance.

Investigating all of these technical approaches would be too broad an undertaking for a doctoral dissertation. Consequently, in this thesis we have chosen to investigate only a system of the last kind, providing the ability to write high-level program designs and then refine them easily into executable programs.

Thus our technical approach is aimed at finding a prototyping a system that is easy to use, yields executable results rapidly, offers expressive power and flexibility, produces executable programs in Ada, and supports programming by progressive refinement.

In considering the above system characteristics it is apparent that any technology that is useful for rapid programming may also be of benefit in Rapid Prototyping. The converse is not necessarily true, however. Our requirement for programming power and flexibility may not always be compatible with good program structure, documentation, and maintainability. Naturally these are important considerations, but they are not as important in

Prototyping as when a program is to be documented and maintained for a long period of time.

SCOPE OF THIS THESIS

This thesis makes two principal contributions. First, it provides concepts, techniques, and a philosophy of Rapid Software Prototyping and characterizes the benefits and limitations of its use. Second, it makes a contribution to programming environments which support Rapid Software Prototyping.

The first part of this thesis establishes what Rapid Prototyping is and shows what can be expected from its use. We claim that Rapid Prototyping is a valuable practice, where appropriate, and has an important role to play in the software lifecycle. Chapter 2 discusses the need for and the usefulness of Rapid Software Prototyping. Chapter 3 is a discussion of prototype design methods, partly in the form of case study examples. The reader wishing to develop a more intuitive concept of software prototyping may find it helpful to read these examples before proceeding with Chapter 2.

The remainder of the thesis is concerned with programming environment issues in support of Rapid Prototyping. Our technical approach is based on using the power and flexibility of Program Design Languages (PDL's)

for both design and implementation. PDL's have been available in the industry for a number of years as semi-formal software design aids and are finding increasingly widespread use [Caine 75, Waugh 80, Hart 81]. Although PDL's resemble programming languages, there is in practice no formal link between the PDL representation of a program and its final implementation. Our approach is to integrate the PDL and the implementation language so that a program can be progressively transformed from one representation to the other. This approach is of particular benefit in prototyping because it minimizes the cost of the transition step from design to implementation. If a small amount of design work is appropriate before launching into the detailed implementation, it can be expressed without digression in the same medium as the rest of the development. If the design itself is quite involved, the system will hold the entire design data base and will manage its progressive refinement into an executable program.

The system also participates in the refinement process by helping the user manage an agenda of the prototype development. This is not a technologically difficult function to perform, but as an interactive aid it is quite helpful for quick assessment of the current status and for planning the remaining work to be done.

These concepts have been implemented in an experimental Rapid Prototype Programming environment. This system supports a language called Castor which is an extension of Ada. The following are the main system features which support the goals described above:

1. Castor calling forms. These are the main syntactic extension of Ada and give Castor its dual nature as a programming language and as a PDL. A calling form is a descriptive program element with parameters and can be used in place of a type, a declaration, a variable name, an expression, or a statement. Refinements are performed by the user to transform a Castor program into an executable Ada program. A calling form may be refined by providing a suitable procedure or function definition, by providing a suitable macro definition, or by replacing it with more detailed program text.

2. Macros. These are used when a refinement is to be given for a calling form used as a type, a declaration, or a variable name. They may also be used for expressions or statements where an Ada function or procedure is not applicable -- for example, if the calling form represents a novel control structure with statements as parameters. A macro facility for a highly structured language such as Ada requires

consideration of a number of issues, including representation of macro definitions, representation of program fragments, and provision for access to the symbol table and the actual parameters supplied in a macro invocation. Macro definitions are written in Castor itself with suitable built-in data types and primitive operations. Program fragments are specified in terms of the source language. This involves some overhead of lexical scanning and parsing during macro expansion but makes it possible to guarantee syntactic validity of the resulting program using limited local checking. This also makes the macro language independent of the underlying representation of programs in the system. A number of special functions are provided for interrogating the symbol table and examining a macro's actual parameters.

3. Representation of refinements. The system employs a mechanism of program attachments for representing the accumulation of multiple program refinements. This makes it possible at any time to look back at the state of the program prior to any given refinement.

4. Definition checking. The system also provides a limited capability for static analysis of the current state of a program for occurrences of undefined calling

forms, variables, types, packages, and exception names. This is for use in obtaining an agenda of work to be done in further refinements.

To give a brief idea of the use of calling forms, let us suppose we wish to write a procedure which operates on a binary tree with an integer at each node. This procedure is to be given a non-empty tree and is to set the value at the root to the maximum value among all the leaves, provided that is greater than the current value at the root. We might write such a procedure as follows:

```
procedure Maxleaf (T: Binary Tree Of (Integer)) is
begin
   For Each Leaf (L) Below (T) Loop
      begin
         if Value At (L) > Value At (T) then
            Value At (T) := Value At (L);
         end if;
      end;
end;
```

This example shows calling forms used in place of a type name, "Binary Tree Of (Integer)," a variable name, "Value At (X)," and a control structure, "For Each Leaf (L) Below (T) Loop begin ... end." The use of calling forms is described further in Chapters 4 and 5.

Chapter 4 is a description of the environment considerations for Rapid Prototype Programming. Calling forms are introduced here along with our notion of

refinement and the concept of program attachments. Chapter 5 is a more detailed discussion of the use of Castor calling forms, particularly pertaining to the definition and expansion of macros. Chapter 6 discusses some of the interactive issues in Rapid Prototype Programming, with particular attention given to the issues of verbosity and clarity in the language. Chapter 7 discusses some of the literature which is related to Rapid Software Prototyping, and Chapter 8 presents conclusions and suggested directions for future work.

Appendix A shows how the principles described in this thesis can be applied in a prototype development of moderate size. The subject is a prototype for an "electronic office" -- that is, a computer system which performs many of the functions of a business office that are traditionally done with paper. This prototype was about a thousand lines long in final form and required an estimated three to four man-weeks for design and implementation.

The Castor system is an extension of an interactive Ada programming environment written at Irvine. This system provides a parser, a pretty printer, and an interpreter for a subset of Ada, as well as a number of other experimental tools. The system and its Castor extensions are written in UCI MLISP, with the Castor extensions consisting of about three thousand lines of MLISP code.

CHAPTER 2

PRINCIPLES OF RAPID PROTOTYPING

"Rapid Prototyping" is a comparatively new term in connection with software engineering. To help establish what it means, it is helpful to compare the concept of software prototyping with prototyping as used in other engineering disciplines.

The prototype of any engineered artifact is a first working model or version of the artifact. It may not have all the polished features that later versions will have, but it is built in order to validate the principles upon which later models will be based. Particular attention is paid to those features whose realization is most _uncertain_ in the final model -- whether it will indeed perform a novel function or will perform a set of known functions in concert.

Frequently in engineering a series of prototypes will be built while refining the technical approach to a problem. Each version suggests improvements which can be made in the next design, either with regard to functional performance or producibility. Usually producibility is a minor

consideration at first, and fabrication of the initial prototypes is quite different from that of the production model.

In software engineering the notion of prototyping takes on a somewhat different meaning. In particular, there is no recurring manufacturing effort in producing a program, since the released version is merely copied as needed. We note in passing that a kind of "recurring manufacturing cost" may be associated with specializing a highly parameterized program (such as an operating system) to various application environments. As a rule, however, this "recurring" cost is very much less than the cost of building the parameterized system in the first place, so the development-intensive nature of software manufacturing still holds. For practical purposes, all the efforts of programming are for a production run of one unit.

To use prototyping in software engineering, we can write a first version of the program, analyze the result, and then write a final version of the program. The cost of writing programs, though, is very high, and it is desirable to save on the costs of the protoype if there is any likelihood that the result will be discarded. As described above, the techniques used in prototyping may differ from standard production techniques. Interestingly though, while engineers may use more expensive, special tooling and

processes in building a prototype, we as programmers wish to do the opposite. Anticipating the possibility of extensive revision, we want our prototypes to be cheap to build and experiment with. Because of this different emphasis, we call our undertaking not just Prototyping, but Rapid Prototyping.

A rapidly generated prototype is not just a first version of the program; it is a cheap version of the program, or "key" parts of it, built for limited experimentation. Because of this the prototype only approximates the production program, and hence there are limits on the inferences that can be made about the final version. If the functional requirements specified by the user are in question, Rapid Prototyping can be most helpful in demonstrating the specified capabilities in a working model. If, on the other hand, the efficient performance of the program is an area of uncertainty, a cheap prototype may not do much to establish the program's feasibility. In this latter case, system analysis or simulation may be required, together with detailed implementation of critical system components. Although such modeling is a kind of prototyping, it is not Rapid Prototyping in the sense we have defined.

We note that a prototype is not to be confused with a mockup. In engineering practice, the latter is a dummy

version which has no functional capability -- for example, a computer cabinet carved out of wood and covered with metallic paint. This can be considered a prototype only in simulating very simple properties of the working article -- physical dimensions, appearance, weight, etc. There is no real correspondence to the mockup in software engineering, except perhaps as a limiting point where all input/output behavior is simulated with stored data and no computation takes place.

MOTIVATIONS FOR RAPID PROTOTYPING

The Software Lifecycle

A number of models have been proposed for the phases a program passes through during its creation and subsequent evolution. It is recognized that these phases are not necessarily strictly disjoint intervals of time. Still they represent qualitatively different activities which all take place. The following set of five stages is typical of models of this sort:

> Requirements Analysis This stage of the program development provides a description of the needs of the user. This concentrates on what functions are to be provided and indicates the kind of a system that will provide them. The requirements form the basis for the detailed specification of the system.

<u>Specification</u> This stage of the program development establishes precise constraints the system is to meet. This provides a description of the precise input/output behavior of the program and may also constrain the program performance and resource utilization. In a real sense, the Specifications constitute a refinement of the Requirements Analysis.

<u>Design</u> The design of a program is a first cut at describing how the program is to perform its work. Major components of the program are identified, major data structures and interfaces are defined, and major algorithms may be specified.

<u>Implementation</u> In this phase, algorithms are provided in a given language for the identified program components so that an executable program is produced. For our present purpose we consider this to include the process of testing, integration, and debugging the program written.

<u>Maintenance</u> This is the conventional name for the remainder of the software lifecycle after program release. The term Evolution seems more appropriate for this phase since, in general, more programming effort is spent on changing the functions of programs than correcting errors [VanHorn 80, Ramamoorthy 79].

Feedback is a very important process in this cycle of development. During any phase it may become necessary to go back and modify decisions which were made at earlier stages. Without realizing this, the above model can be quite misleading. Since users, specifiers, designers, and implementors are each fallible, it may eventually become necessary to revise the work of any of them.

We note in passing that the maintenance phase of a program may involve incremental replay of any or all of the

four developmental activities, depending on the scope of changes that are incorporated. We also note (for "closure") that as the four developmental phases are a general problem solving paradigm, they may be employed recursively for solving any subproblems encountered within the development.

Improved Feedback to Requirements Analysis

Building a software prototype can provide much accelerated feedback to the Requirements Analysis phase of a program development. Since the requirements for a program do not "come from" anywhere, it is very difficult to "check" them to ensure they are right. It is possible, in principle, to demonstrate that an implementation satisfies a design or to show that a design satisfies a set of specifications or requirements. But to see that the initial requirements are "correct," the system behavior must be presented to the user and the user's satisfaction must somehow be measured. In the lifecycle model this is the longest feedback loop in the program development -- from the final program product to the very beginning. Hence this is potentially the most expensive design iteration to engage in.

Rapid Prototyping can provide a means for shortening this feedback cycle greatly. There may not be cause for concern if the application is a familiar one, but for a novel system with a long development time there is a risk of

much wasted effort if the system is not what the user really needs. In a situation involving successive procurement cycles, this may mean much faster convergence to a "satisfactory" solution. In a situation where the development is simply not to be iterated, this may mean a better, more responsive final product, or may even mean the difference between a product which is usable and a product which is not.

Another problem which arises in the procurement of large systems is that requirements may be changed "on the fly," as it were, during the design and implementation phases of the program. Such perturbation can result in cost increases and schedule delays, but this cannot be helped when the alternative is to cancel a multi-year development or to complete it with an unacceptable product. If one or more prototyping phases are performed, it may be possible to stabilize the system goals sooner and protect the final implementation effort from this kind of disruption.

Validation of Novel Designs

Rapid Prototyping can be used to evaluate alternative design approaches while limiting the investment which is risked. This permits a well-founded, objective decision about whether an approach is feasible. It may also provide a quick and handy framework for further tuning.

In this same vein, a prototype program can provide objective evidence of the real bottlenecks in a process by the use of dynamic performance monitoring. Experience has shown that programmers are typically very bad at predicting where the bottlenecks are in their programs [Knuth 71]. Naturally, the conclusions which can be drawn depend on the quality and completeness of the prototype. Measuring the frequency of execution of various program components may give excellent guidance in choosing which functions to implement efficiently.

Used in this manner, Rapid Prototyping provides acceleration of the so-called software learning curve [Boehm 73]. It is well known that program quality is improved and that development cost is greatly diminished when the programmer or programming team has had prior experience building the same program or type of program. This is emphasized by the following aphorism attributed to Ivan Sutherland: "Programs are like waffles -- you should always throw the first one away." This "throw away" approach amounts to an iteration of feedback to the Design Phase of the program, analogous to the feedback to Requirements Analysis described above.

One-Time Applications

Rapid Prototyping techniques can also be envisaged as helping speed up the completion time on "one shot" applications. These are programs which are to be written, run once to get an answer, and then discarded. In such cases, the cost of obtaining the program results may be dominated by the costs of writing and compiling it. In this case it makes no sense to worry much about efficiency, since the increased cost of developing a highly efficient program might far outweigh the cost of running an inefficient one. Rapid Prototyping techniques will be of help in such an application to the extent that they are also rapid programming techniques.

Rapid Prototyping During Maintenance

Rapid Prototyping may also be useful for major program modifications during the maintenance and upgrade phase of the software lifecycle. Maintenance activities frequently involve partial replay of the design processes due to changing system requirements. In fact, the more extensive the modifications, the more the maintenance job constitutes a redoing of the earlier development phases of the program lifecycle.

WHAT TO SACRIFICE FOR RAPID PROTOTYPING

Our objective is to speed up the process of implementing a working program. Since we aim to exceed the productive capacity of conventional software production techniques, we must expect to make some sacrifices while cutting corners. Ultimately we can sacrifice any or all of the following (in order of increasing distance from the final production software):

1. Efficiency
2. Scope of problem size
3. Functional capability

This gives us a rough scale for measuring the degree of approximation of a prototype.

Efficiency

Efficiency is the first thing we think of sacrificing when writing a program quickly. This may mean the heavy use of general procedure calls, ignoring loop optimizations, or use of language features which are convenient to write but inefficient to execute. Efficiency can also be traded off by choosing data structures and algorithms which are easy to describe and understand, resistinw the impulse to use more efficient techniques which are more intricate and demand closer attention to details. If an interactive, interpreted language environment is used instead of a compiled language, this may place limitations on the ultimate run-time

efficiency of the program. We may also use generalized software packages which offer a great savings in programming time, but which may be less efficient than specially coded software.

Scope of Problem Size

Another way to speed development of a working program is to reduce the scope of the problem to be solved. This may make a much simpler approach to the problem feasible, simplifying development but placing the full scale problem out of reach. In such a case, the complexity of the chosen algorithm may become intractable for larger problems, or unoptimized data structures may grow beyond the constraints of the machine size. Great simplification can be achieved if the use of peripheral storage can be eliminated in favor of in-core storage. For example, it is far easier to build a one-pass compiler that builds small procedures in main memory than to build a multi-pass compiler that will optimize and cross reference large programs.

Functional Capability

Another way to save time in writing a program is not to implement everything the program is supposed to do. Instead of just reducing the scale of the problem, we omit parts of the solution altogether -- these are qualitative sacrifices as opposed to the quantitative sacrifices described above.

Naturally if this principle is applied extensively the prototype will not be representative of the final product. The art of engineering prototyping is in implementing just those features which allow resolution of uncertainties in the final product; there must be a purpose for building the prototype, and the functions to be implemented should be chosen with that purpose in mind. The measure of an effective prototype, then, is its ability to run meaningful scenarios of the actual system.

In the Flight Service Station Information System cited in the next chapter, the prototype used canned weather information in place of on-line information, served a reduced number of users, and provided navigational aids for only a limited area of the country. This was sufficient to show what a system was _like_ to use, but could not be used fully by a pilot planning a real trip.

In the next chapter we shall consider this and other examples in greater depth.

CHAPTER 3

METHODS OF RAPID PROTOTYPING

In this chapter we show a number of different approaches which can be used in Rapid Prototyping. We begin with a few examples of the use of Rapid Prototyping in software development. The first example is a prototype of an Automated Flight Service Station. This example shows how system functions can be selectively implemented to get feedback on important issues -- in this case the human engineering of a user interface and the computational load of the basic system operations. The second example is a Custom Microprogram Assembler. This shows how great programming power can be achieved by using existing software in novel ways. The third example is the St. Lawrence Seaway Traffic Control System. This shows how a well-structured program can be significantly abstracted and respecialized for a new purpose. In the remainder of the chapter we shall discuss other techniques for rapid prototype implementation, including the software component approach, program generators, simulation, and reconfigurable test environments.

EXAMPLES OF RAPID PROTOTYPING

Automated Flight Service Station

Our first example concerns automation of the functions provided in a Flight Service Station (FSS). The Federal Aviation Administration (FAA) currently operates Flight Service Stations at airfields across the country for general aviation pilots -- that is, for pilots of private and non-scheduled commercial aircraft. It is here that a pilot gets information he needs for planning a flight and it is here that he files the flight plan for his trip. Necessary information includes current weather conditions, local weather forecasts, general area forecasts, and forecasts of the winds he will encounter while aloft. In addition he may receive briefings on navigation aids which may be out of service or other exceptional flying conditions. A flight plan is filed with the FAA for safety purposes indicating the destination of the flight, the route, and the estimated time of departure and arrival. In the event that the flight does not arrive as scheduled, this information may be used to guide search and rescue operations. The pilot may also be planning a route through a High Density Terminal Area (HDTA). If so then he must also check appropriate restrictions and enter a reservation for this restricted

airspace.

A proposed automation of the FSS functions would help reduce the expense of operating these stations and would help provide quality service to the increasing volume of general aviation traffic. One scheme for partial automation would place a computer terminal in each Service Station for use by the FSS personnel. This terminal would provide access to a nationwide network of computers containing up-to-date information on weather and and flying conditions. In addition, flight plans and HDTA reservations could be entered and would be managed by the computer system.

Complete automation would make the FSS terminal available to pilots on a self-service basis. While very desirable from a cost standpoint, this approach also raises considerations of safety and usability of the terminal by pilots unfamiliar with computer equipment. Care must be taken to check the user's input for consistency and practicality, just as live FSS personnel would. In addition the system must be flexible in recognizing the user's input, give self-explanatory prompting, and print the information it furnishes in a legible format. These are important issues affecting the usability of an automated FSS and are difficult to evaluate in a paper design. This is a case in which a prototype would make it possible to observe the system in action, as it is exercised by a variety of users.

Pilots who are not computer experts cannot be expected to evaluate a written description of a computer system, but given an operational terminal they can objectively evaluate how easy it is to use and whether it serves their needs. The prototype would also show how users learn to use the system and would help pinpoint areas of confusion or ambiguity.

Since an automated FSS terminal puts general computing power at the disposal of the pilot, it is natural to ask if there are other desirable functions which might be provided. For example, commercial airline pilots are provided with a computer-printed flight log prior to flight. The log breaks down the flight into routing segments, giving distances and estimated flight times. Such a flight log must be generated for each trip in order to incorporate current information on the direction and strength of winds aloft. Such a printout could be generated and printed at an automated FSS using known information and would be a great convenience, providing airline-quality information to general aviation pilots. Pilots are also responsible for the weight and balancing of their aircraft, and the automated FSS could also be of assistance in performing these computations.

A prototype of the FSS software described above was actually built in the course of a study for a proposal to the FAA [Taylor 81]. This prototype was built to show the

system functions as they would be seen by a user, without the trouble and expense of supporting real-time weather data or a large number of geographically separated terminals. This prototype showed what a user would do when requesting weather and winds aloft data and when calculating and filing a flight plan. It had commands to make HDTA reservations, generate flight logs, perform weight and balance computations, and send messages to various destinations. It also provided separate modes of interaction for casual users and expert users, and system commands were implemented for entering and updating the system data base.

While providing these important functions and interactive features, the prototype also cut a number of corners. Instead of using on-line weather data, a canned set of data covering a twelve hour period was used. Furthermore, information on airways and navigational aids was restricted to the northeast corridor. Only one user terminal was served, and the entire system was implemented in an interactive language on a timesharing computer system. Since the size of the system data base was reduced, it was possible to place it all in main memory rather than on secondary storage. This simplified the data management aspects of the program considerably. The entire prototype was written in about two man-weeks and was used in a live demonstration at FAA Headquarters in Washington, D.C.

The prototype described here was also used to measure objectively the processing load and program size for each of the functions of the system. Since the prototype was built using an interpreted interactive language, these figures were at best estimates for a production-quality implementation. Still they were objective measures and showed that a custom design would run at least as fast as the prototype and could be made to fit in the same space if necessary.

Care must be taken in extrapolating performance measurements from a prototype to the final system. In the current example one would have to be very careful in predicting the system response characteristics when many terminals are added and when data files are placed on secondary storage. This is to be expected since the prototype was not designed with the purpose of demonstrating these features. A rapidly generated prototype is not built to demonstrate all system features and behaviors at once -- only a full scale implementation can do that. The main purpose of prototyping is to verify responsiveness of a program specification to the real needs of a user. Predicting system performance is only a secondary purpose, and prototyping should not be considered a replacement for careful analysis in this area.

Custom Microprogram Assembler

For our next example we consider the development of an assembler for programs to run on a custom designed micro-processor. We are given that the micro-processor architecture has been defined and that an assembly language is to be specified and implemented. The assembler over its lifetime will be used in the development of a small number of programs, each with a size ranging from about one thousand to four thousand instructions, and the assembler is to be used by a limited community of users. Advanced assembler features are desired, but development costs are to be kept to a minimum.

The approach described below was used to build a practical assembler with important and useful features including macros, conditional assembly, listing control, and cross reference capability [Allen 76]. This entire design and development was made in about one and a half man-months. This includes analysis and design of the language and design and implementation of various semantic checking rules imposed by the architecture of the microprocessor. It was decided to make the language as uniform and high-level as possible, resembling conventional assembly languages, rather than requiring the programmer to specify each instruction field fully. This meant that certain combinations of features had to be forbidden, even though they were

syntactically valid, because they could not be realized by the hardware in a single instruction. Semantic checking was therefore required in the assembler.

The construction of this assembler is described below and illustrates the Rapid Prototyping philosophy of utilizing existing software whenever possible, even in unusual ways not anticipated by the original software developers. In this particular case the assembler continued to be used in its "prototype" form, although modifications and improvements continued to be made over a period of time.

The microprogram assembler was written to take advantage of the features provided by the IBM 370 Operating System Assembler [IBM 72]. Each machine instruction of the micro-processor was defined by a unique 370 Assembler macro definition. Conflicts with the 370 instruction set were eliminated by suppressing all of the 370 machine instruction mnemonics. Semantic checking and instruction formatting were accomplished by a few system macros which were heavily parameterized and were called by the individual micro-processor "instruction" macros. In the end, each instruction was assembled as a "define constant" command to the 370 Assembler.

It was necessary to put the assembled object program into a form suitable for loading into the Read Only Memory (ROM) of the micro-processor. Having used the existing 370

software to generate the object program, it was most natural to use the existing 370 loader to handle reading the relatively complex 370 object program format. Hence the object code was simply loaded into memory together with a specially written program called the "postpass." This program was written to produce the desired object program format directly from the in-core memory image.

The postpass had one other function which could not be performed by any exisitng software -- reading the 370 assembly listing. It was necessary to edit the listing to remove the "define constant" statements generated by the macros, while retaining the printed value of each assembled instruction. It was also necessary to change the addresses displayed from byte addresses (the 370 address space) to word addresses (the micro-processor address space). In addition to these functions, the postpass also performed simple clerical functions such as re-paginating the listing and filling out the object program to the size of the Read Only Memory chips.

This example illustrates a way in which a prototyping effort can take advantage of existing software. In this case, there were existing assembler functions already available for parsing, symbol table maintenance, code generation, macros, conditional assembly, and cross referencing. These were coupled with a meta language (370

macros and conditional assembly language) in which it was possible and convenient to express the logic of the custom application. Because of this there was no need to deal with the internal program structure or data formats of the 370 assembler -- the "borrowed" software was utilized as a whole. It was necessary, however, to write a small, low-level program, the postpass, but the cost of this plus the macro definitions was minute in comparison with the cost of writing such a system from scratch.

St. Lawrence Seaway Traffic Control System

This is an example of a prototyped system based on existing software by using selected internal modifications. The original software was a computer graphics program for simulating a radar air-traffic control system. A video display was used to show moving aircraft symbols. These symbols, with associated description data blocks, were maintained in position on a background map of the airspace. The movements of the aircraft were modeled and displayed to simulate the behavior of an entire air-traffic control system.

A new application was proposed for the needs of the managers of the St. Lawrence Seaway. A capability was needed to help monitor and control the shipping traffic on the Seaway [Taylor 81]. Because of the good modularization of the air-traffic simulation program, it was possible in a

few days to modify it to simulate a sea-traffic control system. The time and distance scales of the system were changed and equations for ship motion were substituted for the aircraft modeling equations. Display symbols were changed to indicate ships and the background map was changed to show the geography of the seaway. This is an application that was not forseen by the designers of the air-traffic simulator, but because the system was well modularized and parameterized it was comparatively easy to reuse the general structure of the program with new parameter values, table contents, and selected subroutine definitions.

METHODS OF RAPID PROTOTYPING

Reduction of Scope

The examples given above illustrate several different methods of Rapid Prototype implementation. The first of these is selective choice of just how much to implement. Naturally this choice depends on the reasons for building the prototype in the first place. In the Automated FSS example, the primary purpose of the prototype was to demonstrate the interactions between the user and an FSS terminal. The judgement was made that the system response time to requests was not a crucial factor to model and verify. It was therefore unnecessary to simulate

competition among independent terminals, and a prototype serving one terminal was therefore sufficient. In addition, the purpose of this prototype could also be satisfied by using canned weather data rather than real time data and by restricting the system to a limited region of travel.

Another common capability which can sometimes be bypassed in a prototype system is detailed error handling and error recovery. Naturally in some systems this will be an important consideration, particularly if the error handling has a great impact on the user interface. In many cases, though, error analysis and handling are best postponed until the main functional cases are fully explored and understood. Where canned data replaces real-time or user-supplied data, it may be convenient to skip checking that would be necessary in the full operational system.

Again like the FSS example, it makes sense to limit the size of data structures so that they will fit into main memory. This not only simplifies the writing of the prototype but also makes it much more flexible with respect to algorithmic or functional changes. By taking advantage of the random access properties of the computer memory it is possible to experiment with different algorithms which would require complicated redesign of disk or tape handling algorithms.

VHLL's and Program Generators

The Microprogram Assembler example is an instance of using a very high level programming capability. Viewing a general purpose assembler with macros as a meta-assembler (that is, an assembler capable of being specialized to any of several assembly languages), the job of implementing a given assembler becomes a programming task in a very specialized higher level language. Similar capability is provided by a compiler compiler [Brooker 63]. Other related systems are business oriented program generators such as BDL [Goldberg 75] or PROTOSYSTEM I [Martin 74]. Very high level languages for general purpose use have also been developed, such as SETL [Dewar 78], VERS2 [Earley 74, Earley 75], and MADCAP VI [Wells 72].

Reusable Software

Another approach to Rapid Prototyping is to adapt or use parts of programs already written, as in the St. Lawrence Seaway example. The effectiveness of this method depends greatly on the flexibility of the program structure and the quality of its documentation. A program necessarily contains assumptions about the ways in which it may be changed. Unfortunately it is never possible to anticipate all of the ways in which a program may be changed or generalized, but good programming practice demands some consideration of this issue during design and

implementation.

The Component Software approach is a formalization of the process of reusing software [Neighbors 81]. In this approach there is a large preliminary effort made in preparing reusable fragments for a particular programming domain. Also associated with a domain are parsing rules, optimizing program transformations, and pretty-printing rules. Operations in one domain are expanded in terms of operations in other domains, with transformation rules being applied to simplify the resulting program. For this approach to be effective, a considerable effort must be expended to define the entities and operations in a domain. Included in the analysis of the entities and operators of a domain is the assumption that the operations so defined are the ones which are to be re-used in different contexts. The power of the method lies in the use of the transformation system to integrate the assembled program fragments (which are typically rather small) and to customize the resulting system.

Simulation

Although not exactly a prototyping technique, simulation is an analysis technique which can be used for some of the same purposes. This is particularly so when the effect of an algorithm cannot be analytically predicted. Consider for example a proposal for unified traffic signal

control in a large metropolitan area. A prototype to control a subregion of the entire area would not be particularly beneficial. The technology for monitoring traffic and controlling intersections is well established, and the novel aspect of this proposal is the centralized nature of the control. The usefulness of such a global scheme would best be demonstrated by a computer simulation of the system, using randomly generated or previously recorded data.

Simulation is more for design validation than requirements validation. Simulation and prototyping share the same need for careful planning and the same problems if critical system considerations are misunderstood or ignored.

Reconfigurable Test Environments

"Embedded systems" are computer applications in which the computer acts as only one of many integrated system components. Development of the software for such a system generally takes place in an artificial environment where external inputs and controls can be easily simulated and monitored while exercising the embedded computer software. Thus the software for a torpedo, a satellite, an air-traffic control system, or a hospital patient monitoring system is fully exercised in a laboratory environment before being turned loose in the real world.

To accomplish Rapid Prototyping of an embedded system it is important to be able to configure a test environment rapidly as well. Just as tools for developing software systems come in families depending on the general nature of the task, it also makes sense for a general application area to have reconfigurable hardware and associated software to support testing. Examples of typical capabilities are clocks, radar or sonar sensor inputs, gyroscope or accelerometer inputs, facilities for collection of system performance data, and data analysis tools.

RAPID PROTOTYPE PROGRAMMING

The methods of software prototyping described above are case studies of techniques which may or may not be applicable in any given situation. For our theory of Rapid Prototyping to be practical we also need techniques which can be applied more generally. In the following chapters of this thesis we develop concepts for a programming environment for Rapid Prototype Programming. The purpose of such an environment is to facilitate the rapid, high-level description of a program followed by its refinement into an executable representation. This environment can be used to help develop programs quickly, incorporating where possible the techniques described in this chapter.

CHAPTER 4

RAPID PROTOTYPE PROGRAMMING ENVIRONMENT

In this chapter we consider how a programming environment may support the activity of Rapid Prototyping. This includes both language features and mechanical tools for rapid development of prototype programs. We note that the techniques described here may also apply to some extent to rapid development of any software -- a goal we may call Rapid Programming. In order to emphasize that our concerns are more limited, and that where necessary we are willing to sacrifice program efficiency for ease of development, we call the object of our investigation Rapid Prototype Programming.

We have two main goals in a Rapid Prototype Programming environment. Our first is the rapid and convenient expression of what a program is to do. The second is to facilitate changing implementation decisions as the programmer's approach to the problem evolves. We support programming by refinement. By this we mean that the programmer at first expresses his program as a high-level description which is free from commitment to detailed

decisions. The programmer subsequently refines the meaning of various parts of the program, interpreting them in terms of known language features or in terms of still other abstractions which will themselves be refined in time. By keeping a history of these developments, the system can assist the user in retracing and revising the steps from any given point in the development. This approach shares aspects of philosophy with the Harvard Program Development System [Cheatham 79].

THE TARGET LANGUAGE, ADA

A software prototyping facility must produce programs in an executable language, and for our discussion of prototyping we shall use Ada as our target language[Ada 80]. While appropriate for systems programming and for "embedded" computer applications, Ada is, in fact, a general purpose implementation language and shows promise for widespread use. In addition to giving us a real and practical context in which to explore prototyping, this choice also gives us the opportunity to evaluate this new language from a novel and important point of view.

We note that there is also current active interest and research into the development of integrated programming environments for Ada [Stoneman 80, Standish 80, UCI Workshop 78]. These environments are unified

collections of tools for the development and maintenance of software. In this setting we propose that a Rapid Prototyping facility may be a valuable tool for programming environments of the future and should be integrated within such a system to take advantage of the presence of other tools.

The choice of Ada as a target language quickly brings up important issues about the particular requirements for software prototyping. The design goals of Ada and the goals of Rapid Prototyping are somewhat at odds with each other since they emphasize the concerns of different phases of the software lifecycle. In the design of Ada, emphasis was placed on the long-term life of programs and the legibility of programs for documentation and maintenance. For this reason sacrifices were made in the compactness of the language and the ease of writing an initial program. The programmer must specify a great deal in writing a program, sometimes with considerable redundancy for both visual and mechanical program checking.

For prototyping, on the other hand, compactness and ease of expression are at a premium. We want to write concise programs which are free from redundancy and low-level implementation details. Redundancy is particularly to be avoided since we want a medium in which decisions can be expressed and changed easily.

These conflicting needs for completeness and brevity cannot be met simultaneously. We take the view that an explicit mechanical conversion is required from the prototype form to the executable form. Such a transformation, performed by an "Ada laundry" process, allows a prototyper to express himself in a streamlined language, while allowing for conversion to the pure language at a later time. The only feature of this kind implemented in Castor is the allowance made for omitting the redundancy of package specifications. A number of other concepts for an Ada laundry are proposed in Chapter 8.

Another important transformation is from the prototype to the final production-quality implementation. As we have seen, these two representations may address radically different forms of the problem or approaches to its solution. Still the use of the same target language allows the implementors to incorporate portions of the prototype whenever this is possible. In addition, it may be possible to take advantage of program fragments generated from library components in the prototyping system. The same programmers should be involved both in the prototyping effort and in the final design and implementation. This gives the implementors the significant advantage of prior experience with the problem and approaches to its solution [Boehm 73].

PROTOTYPE DEVELOPMENT IN CASTOR

In writing prototypes we shall use a language extended from Ada which we call _Castor_. (This name derives from _Castor canadensis_, the scientific name for that indefatigable species of architect and engineer, the North American beaver.) We shall also refer collectively to the program manipulation functions which handle Castor programs as the _Aide_. As mentioned above, one of the purposes of the Aide is to "launder" Castor programs, removing those liberties which have been taken with the rules of "pure" Ada. Another important function of the Aide is to manage and to assist the development of Castor programs by refinement from very abstract program descriptions to fully executable programs.

Our view of a programming environment is that programs may be entered and output as text files, but within the system a program is represented in a structured internal form. This structure is tree-like and reflects the syntactic phrase structure of the language imposed by the Reference Grammar [Ada 80]. We find it desirable, however, to keep the details of this representation hidden from the user as much as possible. The user only sees and expresses program fragments in the source language. This means that the user is spared having to learn a new language and the

mapping between it and the external program form. This also means that the system/user interface can be common among systems using different choices of internal representation.

An integral component of the Aide is a structure-oriented editor by which the user may enter, modify, and refine Castor programs. Because the editor knows the syntax of the language and maintains programs in their internal form, it can maintain their syntactic validity. By appropriate prompting and checking this can give the user immediate feedback on syntactic errors and gives him a higher conceptual level of discourse for dealing with his program. Since this kind of editing facility has been described elsewhere [Feiler 79, Teitelbaum 81], our main concern in this thesis is to describe the program manipulation concepts particular to Rapid Prototyping.

SWITCHING CONTEXTS DURING PROBLEM SOLVING

Calling Forms in Castor

An important capability in developing a program is to be able to switch easily from consideration of one problem to consideration of related problems. In order to encourage and facilitate this we introduce Castor *calling forms*. In use, a calling form is written much like a procedure call and acts as a self-documenting description of what the call

does. Briefly stated, a calling form is written as a sequence of identifiers with interspersed parenthesized parameters -- for example:

```
Find The Deepest Leaf (L) In Tree (T);
```

In programming by refinement a programmer writes a solution assuming the availability of procedures not written yet. Later, definitions of these procedures are written which may in turn use still other unwritten procedures. This process continues until all needed procedures have been written. This process is also called "top-down programming."

Virtually all high-level languages, including Ada, support the use of procedures in this manner, allowing the program structure itself to reflect the development process. This principle of development can be applied to other aspects of the program as well. Most high-level languages also have function calls -- these are used to represent value computations which are defined in a textually distinct part of the program. Pascal and Ada also allow the naming of data types which are remotely defined, and Ada has a limited capability for parameterizing references to such types.

In Castor we permit the use of calling forms as statements (in which case they act as procedure calls) and

as expressions (in which case they act as function calls). In addition we can also use them as declarations (of objects, types, subprograms, and so on), as types (in defining type identifiers, constants, variables, and parameters), as variables names, and as control structures. The various uses of calling forms in Castor are described in more detail in Chapter 5.

The premise of prototyping is that at times it is best to bypass details in doing a job in order to get some kind of "finished product," however preliminary. The calling form is a way for the programmer to avoid digressing into details so that he can finish a chain of reasoning or description at a given level of discourse. Used in this way a calling form can serve as a brief description and a reminder of what needs to be done later. This serves a dual purpose as the name of the subprogram and as an in-line, self-documenting description of the function to be performed. A calling form may later require a full definition of details by the programmer, or such a definition may be invoked from a library of definitions known to the Aide.

Aide Management of Calling Forms

During development, a program may have a number of calling form references which need to be refined. Known to the Aide, these constitute a formal agenda of the work that

remains to be done. At any time the Aide can report on the status of various modules of the program under development and can automatically prompt the programmer to choose a new, unimplemented calling form to define as successive refinements are completed.

The Aide can also keep track of the points of use of each calling form. For a calling form yet to be defined this helps the programmer remember the context and requirements for the new definition. For both defined and undefined forms this is useful for reviewing the contexts of invocation and for revising these contexts or calls when necessary.

Sometimes it is desirable instead to write the definitions of calling forms before writing the components that use them. Applied consistently, this is the "bottom-up" approach to programming. This can also be done using the Aide, and the Aide can be of some assistance in showing those calling forms that have been defined but which have not yet been used. Merely using the defined calling forms at least once does not provide a precise measure of progress toward the final program goal, however. Even if all calling forms have been used at least once, this does not mean that the program is finished, and there is no indication of what needs to be done next. Bottom-up programming does not give the Aide quite as much opportunity

to assist, therefore, since the Aide has less information about the programmer's plans and what remains to be done.

These features of the Aide help the programmer switch from one problem to another. The use of calling forms helps the programmer to manage the development of his program, helps him articulate a given line of reasoning quickly without digressing into unnecessary details, and provides a formal agenda and framework for switching among the many problems he must eventually solve.

PROGRAMMING BY REFINEMENT

There are two ways in which a user may wish to define a calling form. The first is to update the program text at the point where the calling form is used -- this is most appropriate when the calling form is used only once. The second is to provide a definition elsewhere and leave the calling form invocation as a reference through a suitable symbol table mechanism. We call the first method _in-place refinement_ and the second _remote refinement_. We shall discuss in-place refinement in the remainder of this chapter and shall describe Castor features for remote refinement in Chapter 5.

When a calling form is refined in-place, we do not want to lose the unrefined text of the calling form. The calling form can still act as a compact description of the refined

program text. Furthermore, at some time in the future the user may wish to undo the refinement. It is also desirable to be able to view the program in its initial unrefined form as a form of program documentation.

An in-place calling form refinement can be displayed in the following manner:

```
--(A2)  Merge Elements (X1,X2) In Partition (P);
declare
   Y: Set Of (T);
begin
   Y := Union (X1,X2);
   Remove (X1) From (P);
   Remove (X2) From (P);
   Add (Y) To (P);
end;
```

In this example a statement operating on a set partition is refined so that the same action is expressed in terms of basic set operations. Since a partition can be implemented as a set whose elements are sets, the merging of two partition elements can be expanded as the computation of a new element value, Y, which is the set union of the given elements, X1 and X2.

The first line appears as a comment and contains the unrefined calling form. This comment represents an entity managed by the Aide called a program attachment. These are described later in this chapter.

The Aide supports the notion that refinements are grouped together. The term "A2" in the program attachment

is the name of such a refinement group -- this name is simply an arbitrary identifier. In this example of refinement, the decision to represent a partition in terms of set primitives motivated the refinement shown. In a real program we would naturally expect this decision to influence the refinement of other parts of the program as well. The places referring to P might include the declaration of P, the point where P is initialized, and points where P is examined or modified. The name of the group may be used to sequence through the refinements for listing or editing, and it may also be used to undo the refinements all at once, if necessary.

Frequently, refinement groups are independent of each other and can be introduced and possibly removed in any order. For each refinement in a group the Aide replaces the old program fragment (a calling form) with a new program fragment and associates the old unrefined fragment with the new fragment by means of a program attachment.

The programmer will naturally introduce the refinements of his program in some order. It may happen that later refinements depend on refinements which have already been made. For example, one might further refine the partition example above by a group, B2, giving a linked-list implementation for sets. Thus every point refined by A2 to implement partitions with sets will now be further refined

by B2 to implement those set operations in terms of other primitives. If the programmer wishes to undo the A2 group, this naturally entails the undoing of all of B2 as well.

It can also be that two refinement groups interact without either being totally dependent on the other. For example, suppose the partition P discussed above were declared and used as follows:

```
--(A2)  P: Partition Of (T);
P: Set Of (Set Of (T));

. . .

Find Arbitrary (X) In (P);
if (A) In (X) then
   . . .
end if;
```

Note that refinement group A2 has refined the declaration from a partition to a set of sets. Suppose we now wish to refine T to be a limited range of integers, 1 to N. We then get the following:

```
--(B2)
type T is 1 .. N;

. . .

--(A2)  P: Partition Of (T);
--(B2)  P: Set Of (Set Of (T));
P:  Set Of (array (T) of boolean);

. . .

Find Arbitrary (X) In (P);
if X(A) then              --("(" B2)  (A) In (X)
   . . .
end if;
```

In this version of the program we note that the declaration of T has been introduced by group B2. Rather than being a refinement of a calling form, it has simply been introduced out of nowhere. The declaration of P has been refined again to permit the use of boolean arrays to represent sets of elements of T. The *if* condition "X(A)" now replaces the previous calling form "(A) In (X)," meaning that this test is now accomplished by using A to index the appropriate boolean in the array X. The comment

```
--("(" B2)  (A) In (X)
```

indicates a program attachment which is attached to the expression "X(A)" rather than the whole *if* statement. Other ways of specifying program attachments are given below.

The point we wish to make here is that neither A2 nor B2 is strictly dependent on the other, although they do interact. If the programmer decides to undo A2 it is necessary to undo those refinements of B2 which are dependent on refinements of A2. Nevertheless it may be meaningful and desirable to retain those parts of B2 which are not dependent on A2. In this example, the declaration line would have to be completely undone back to its original form:

```
P: Partition Of (T);
```

but the declaration of T might still be retained:

```
-- (B2)
type T is 1 .. N;
```

Note that the *if* condition "X(A)" is well-defined if X is known to be an array, but this is only known if the B2 declaration is in force. Hence this refinement should be undone too. For this reason, the Aide will expect to undo all refinements of B2 if any refinement of B2 is dependent on A2. However, the user may also use the Aide to examine the refinements of B2 which are independent of A2 on an individual basis and remove them selectively instead.

We wish to retain the refinements of an undone refinement group, just in case the programmer again changes his mind. Undone refinements are also represented as program attachments. An undone refinement is displayed as follows:

```
--(Undo B2)  P: Set Of (array (1..n) of boolean);
--(Undo A2)  P: Set Of (Set Of (T));
P: Partition Of (T);
```

Program Attachments

A *program attachment* is much like a comment in that it contains information distinct from the program text which is associated with a specified portion of the program. The presence of a program attachment is generally ignored except

under specified circumstances. Comments themselves are a kind of attachment whose only form of "recognition" is to be printed in listings of the program. The concept of an attachment is more general, though, since it is associated with a fragment of the program, rather than just a point in the text: in general an attachment is associated with a phrase or sequence of phrases in the program, as defined by the grammar of the language. Program attachments are a powerful programming environment concept and can be used for a variety of purposes, including software development status, program measurement counters and breakpoints, update-version information, and comments intended for different audiences or points of view [Standish 80].

Since we have stipulated that the user interfaces with the system only in terms of the source language, it is necessary to represent in source language where an attachment is made as well as its value. We therefore adopt conventions for representing program attachments as comments in the source language form of a program. In this way attachments can be interactively displayed in source language terms and can be stored in a normal source program file to be re-read later.

In Ada a comment appears at the end of a text line (which may be otherwise empty), beginning with two hyphens, "--." For example,

```
A := B;        -- This is a comment about A:=B
```

We use the convention that a left parenthesis or a number following the two hyphens distinguishes a program attachment from a regular comment. Text within the parentheses identifies the kind of attachment and the information being attached. A refinement, for example, is represented by attaching the old calling form to the refined program text. The following is such a refinement:

```
--2(A1)  Set (X) To The Maximum Value Among (X,Y,Z)
if Y > X then X := Y;
end if;
if Z > X then X := Z;
end if;
```

The number "2" indicates that the attachment is being made to the two statements which follow. If this number is omitted, the default is one, as in our previous examples.

We have noted that a refinement may be introduced from nowhere, as with

```
--(B2)
type T is 1..n;
```

When this is undone, the attachment must remain as a place holder attached to no statements. A "group" of zero statements serves this purpose:

```
--0(Undo B2)  type T is 1..n;
```

We make the convention that an attachment is continued to another line if the following line is empty except for a comment beginning with three hyphens. For example:

```
--(B2) Find The Root Mean Square Noise Level Of The
---    Current Sampling Interval
R := RMS_Noise(I);
```

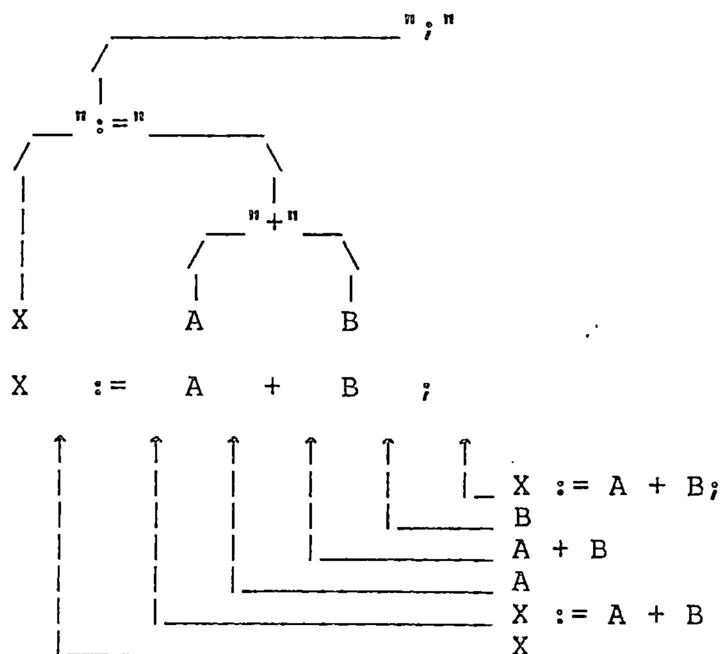
We have seen at least two kinds of attachment: refinements and undone refinements. In general an attachment consists of the information to be attached, an attachment type, and one or more parameters associated with the type. For example, a refinement has the refinement type "Refinement" and one parameter which is the group name. We can write an attachment generally as

```
--(attachment_type param_1 param_2 ...)  value
```

When the attachment_type is omitted, it is assumed to be "Refinement."

The convention described above is convenient for attaching to phrases in the language which are written on successive lines. These include statements, declarations, the alternatives in a "case" statement, and others. For smaller phrases we must show not only the position but also the size of the phrase. One possible solution is to place directions in the comment which specify the beginning and end of the phrase. A more concise convention can be

established, however, using the position of the comment in the text. The motivation for this correspondence can be seen in the following figure:



In the center of this figure vertically is the statement "X:=A+B;." Above this statement is a parse tree showing the operator symbols "+," ":=," and ";" as internal nodes of the tree and the operands as leaves. Below the statement are indicators showing each possible point where a comment might be inserted. For each such point a phrase of the statement is identified. We note that each possible comment location can be identified with exactly one subtree of this tree and that each subtree is represented once. The convention used is that we examine the token immediately to the left of the comment locus. The designated subtree is the subtree having

that token at its root. If the token is a leaf such as "X," then the subtree naturally has only one element, the leaf itself.

It is necessary to define the phrases of the language. The most natural definition is to use the reference grammar and say that a phrase is any string derived from a single nonterminal of this grammar [Ada 80].

We determine what phrase an attachment is attached to by the following steps:

1. Determine the terminal symbol, T, in the program which is associated with the attachment.

2. Let "N -> rhs" be the production in the parse of the program where T is generated -- i.e., T appears in rhs.

3. Then the desired phrase is the terminal string generated in the program by this instance of N.

We illustrate this convention by the following example:

```
1)  if i > j then
2)      A := B;
3)  else
4)      X := Y;
5)  end if;
```

The following table shows which phrases are selected by which terminal symbols. (Terminals appearing in more than one line are distinguished by their line number.)

nonterminal	phrase selected
_if_(1)	_if_ i>j _then_ ... _end_ _if_;
then	_if_ i>j _then_ ... _end_ _if_;
else	_if_ i>j _then_ ... _end_ _if_;
end	_if_ i>j _then_ ... _end_ _if_;
_if_(5)	_if_ i>j _then_ ... _end_ _if_;
";"(5)	_if_ i>j _then_ ... _end_ _if_;
">"	i>j
":="(2)	A:=B;
";"(2)	A:=B;
":="(4)	X:=Y;
";"(4)	X:=Y;

(Note that the introductory parse tree given above to motivate this development does not follow this rule precisely since it shows ":=" and ";" on different internal nodes in the tree.)

The most natural token for a comment to refer to is the token immediately before the comment. This works out nicely for structured statements. In the following, the comment refers to the whole _if_ statement:

```
if X > M then        -- Assure current maximum is valid
   M := X;
end if;
```

Similarly, it is easy to attach to other structured statements at their top level:

```
while P loop         -- attachment to while loop
   S;
end loop;
```

```
case i is                -- attachment to case statement
   when 1 => S1;
   when 2 => S2;
end;

begin                    -- attachment to this block
   S1;
   S2;
end;
```

One-line statements, in particular, include semicolons in their top-level syntax:

```
A := B;                  -- attachment to assignment
return 5;                -- attachment to return statement
exit cycle when P;       -- attachment to exit statement
```

This simple method is not ideal in all cases, however. Expressions and parameter lists, for example, can become hard to read:

```
A := B * (C +          -- attachment to C+D
     D) * (E -         -- attachment to E-F
     F);
```

or,

```
Procl ( A +            -- attachment to A+5
     5, B              -- attachment to B
     , C +             -- attachment to C+D
     D);
```

For this reason we add the ability to refer to a token within the line by placing it in string quotes:

```
A := B*(C+D)*(E-F); --("+" A2) ..something about C+D
                    --("-" A2) ..something about E-F
```

Of course instances of the desired token may appear more than once in a line. A number parameter, as for statements above, is used for this:

```
P(A+5,B,C+D);     --("+" A2)  ..something about A+5
                  --("B" A2)  ..something about B
                  --2("+" A2) ..something about C+D
```

Using these conventions we can therefore make an attachment to any phrase in a program or to groups of consecutive phrases representing consecutive declarations or statements. In most cases simply the location of the comment is sufficient to identify the phrase being attached to. In the prototype described in the appendix, attachments to statements and declarations were found to be much more common that attachments to expressions. Nevertheless, facilities for expressions were also found to be necessary.

AN EXAMPLE: THE EIGHT QUEENS PROBLEM

In concluding this chapter we take a scenario from the literature which illustrates development of a program by stepwise refinement. This is a program described by Wirth to solve the Eight Queens Problem [Wirth 71]. In this paper Wirth used a variant of Pascal notation to develop his solution. The high-level program, transliterated into Ada calling forms, is as follows:

```
1)  procedure Eightq is
2)     Board: <>;
3)     Pointer: <>;
4)     Safe: <>;
5)  begin
6)     Consider First Column;
7)     loop
8)        Try Column;
9)        if Safe then
10)          Set Queen;
11)          Consider Next Column;
12)       else
13)          Regress;
14)       end if;
15)       exit when Last Col Done
16)          or Regress Out Of First Col;
17)    end loop;
18) end;
```

In Castor we have extended the syntax of variable declarations to allow the characters "<>" to be used in place of a data type. This permits the programmer to identify the variable, as in lines 2, 3, and 4 above, without deciding on a specific data type. We note that a user may be similarly unspecific with data type definitions and constant values.

At this point the user may wish a summary of what remains to be specified, and the Aide prints the following list:

```
type of Board
type of Pointer
type of Safe
procedure Consider First Column
procedure Try Column
procedure Set Queen
procedure Consider Next Column
procedure Regress
function Last Col Done
function Regress Out Of First Col
```

Following Wirth's development, the programmer's next step is to refine the procedure "Try Column," which he does as follows:

```
procedure Try Column is
begin
   loop
      Advance Pointer;
      Test Square;
      exit when Safe or Last Square;
   end loop;
end;
```

Assuming the user chooses to make this refinement in-line, the programmer would now look as follows (choosing to name this refinement "R1"):

```
1)   procedure Eightq is
2)      Board: <>;
3)      Pointer: <>;
4)      Safe: <>;
5)   begin
6)      Consider First Column;
7)      loop
8)         --(R1) Try Column;
8.1)       loop
8.2)          Advance Pointer;
8.3)          Test Square;
8.4)          exit when Safe or Last Square;
8.5)       end loop;
8.6)       --
9)         if Safe then
10)           Set Queen;
11)           Consider Next Column;
12)        else
13)           Regress;
14)        end if;
15)        exit when Last Col Done
16)           or Regress Out Of First Col;
17)     end loop;
18)  end;
```

We note that at any time the Aide may display the current form of the program in three ways with respect to this refinement R1 (with the same choices independently available with respect to other refinements as well):

1. Show the program prior to R1.

2. Show the program with R1 in effect, together with an indication of the unrefined version of the program, using program attachments (as in the print-out above).

3. Show the program with R1 in effect, without any display of its unrefined form.

At this point the user's request for status would no longer show the need to furnish "Try Column," but would show the following new items:

procedure Advance Pointer
procedure Test Square
function Last Square

At this point the user would pick another unresolved item and would proceed as above. We note that this particular program is evidently a highly tuned example of refinement, anticipating in some unrepresented way the exact refinements that will take place later. Nevertheless, this example serves quite well to illustrate this family of functions in the Aide. A more extensive example may be found in Appendix A.

CHAPTER 5

THE USE OF CALLING FORMS

CALLING FORMS

Our basic method of extending the Ada language is by the use of program *calling forms*. This name reflects their origin as a general notation for procedure and function calls. We will also use the term "calling form" to refer to analogous extensions in other parts of the language.

The basic calling form consists of a string of identifiers, separated by spaces if necessary, with parameters interspersed. A calling form may begin with parameters or identifiers and may likewise end with either parameters or identifiers. If a calling form has no parameters then it must have two or more identifiers to be distinguished from an ordinary Ada identifier.

Our notation is motivated by the parameter commenting convention of Algol 60 [Naur 63]. In this language a procedure call could be written:

```
MOVE (A, B, C)
```

or:

```
MOVE(A) FROM:(B) TO:(C)
```

In this latter form the strings ")FROM:(" and ")TO:(" are defined in Algol to be syntactically equivalent to the commas in the first call. The character strings "FROM" and "TO" are treated as comments and are not checked for consistent use with the procedure definition or among different calls.

For Castor calling forms we consider all the identifiers to be significant -- together they constitute the name of the calling form. In our notation the procedure call above would be written:

```
Move (A) From (B) To (C);
```

This is defined to be equivalent to the following pure Ada statement:

```
Move_From_To (A, B, C);
```

Parameters to a calling form may be names, expressions, or nested calling forms. In certain cases they may also be Ada declarations, discrete ranges, or lists of statements. When the parameters are statements, they are set off with "begin ... end" rather than ordinary parentheses -- this is for the sake of appearance since this is how statements are

grouped in other parts of the language. We note particularly that the statements are not necessarily executed in the sequence shown -- they just form a list which can be disassembled under the direction of a suitable macro definition. This is a possible drawback, since in other parts of the language such statements are always executed in sequence. A possible modification of this convention might be to introduce one or two new reserved words for this purpose.

The following are examples of legal calling forms:

```
1)  Move (A) On Board (B) To Position (X+1,Y+2)
2)  (X) Is A (Set Of (Integer range 1..10))
3)  Maximum Execution Time Of
        begin
           P1(X);
           P2(X);
           P3(X);
        end
4)  (N) Factorial
5)  The Last Leat In Level Order
```

Lexical Considerations

We note that in the examples above a number of the calling form identifiers are also Ada reserved words ("Is," "Of," "In"). These and other reserved words are necessary if writing of calling forms is to be at all natural. One way to make these available would be to allow reserved words as identifiers, but only within multi-identifier calling forms. This rule gives us the closest possible

compatibility with pure Ada but causes some difficulty in parsing. For a pathological example, consider the following two statements:

```
1)  begin                                              5-1)
2)     for I in Integer loop
3)        begin
4)           . . .
5)        end;
6)     end loop;
7)  end;

8)  begin                                              5-2)
9)     For I In Integer Loop
10)       begin
11)          . . .
12)       end;
13) end;
```

Statement 5-1 contains an Ada for loop comprising lines 2 through 6, and statement 5-2 contains a calling form comprising lines 9 through 12. If we allow reserved words in calling forms, the parser can only distinguish 5-2 from 5-1 by finding the end of line 13 before finding an end loop (line 6). Note that there might be an arbitrarily large number of statements to parse after line 5 (or 12) before that determination could be made. In the first case these statements would belong to the for loop, but in the second case they would belong to the enclosing begin block. Since the parsing handle cannot be identified with any bounded lookahead, the grammar cannot be LR(k) for any k.

An expedient was chosen in Castor which requires bending one of the rules of Ada -- namely, the rule that

upper case and lower case letters in identifiers are not distinguished. Instead, we require reserved words to be written all in capitals; regular identifiers may contain upper and lower case letters, except that an identifier spelled the same as a reserved word must contain at least one lower case letter. A rule such as this naturally causes some compatibility problems, but we note the following:

1. To read a valid Ada program into Castor it is only necessary to capitalize all letters in identifiers. A switch setting causes Castor to do this, and hence Castor can read any valid Ada program.

2. To read a Castor program into another Ada system, we will have no problems provided that the only use of reserved words is in multiple-identifier calling forms. When the program is made into legal Ada, calling form identifiers will either disappear during macro expansion or will be merged into legal identifiers, such as:

   The_Last_Leaf_In_Level_Order

This method of distinguishing reserved words was arbitrarily chosen, and a number of alternatives might have served equally well. In the body of this thesis, reserved words are showed underlined (<u>for</u>, etc.), while in the Castor

transcripts in the appendix reserved words are shown capitalized as they appear in actual use (FOR, etc.).

The only other modification of the Ada lexical rules is the introduction of one additional reserved word, _macro_, as described later in this chapter.

Subprogram Calling Forms

There are different ways to interpret calling forms used in a prototype program. As we saw in Chapter 4, the calling form may disappear altogether through the process of refinement before the program is ever executed. The simplest way to interpret the calling form, leaving it unchanged, is to use it as a straightforward Ada _subprogram invocation_. A more powerful method is to give a macro definition which computes a program fragment to replace the calling form. This makes calling forms a tool for _language extension_. Still greater power can be achieved by providing macro and subprogram packages for different application areas. We shall now consider these methods in greater detail.

We can interpret an executable calling form as a call to a procedure or function subprogram. The identifiers of the calling form together constitute the name of the subprogram. In this primitive application, the user writes a calling form as a descriptive name for a subprogram which he will write in detail later. The following restrictions

apply:

1. Subprogram calling forms may only appear in executable statements or expressions. Objects, types, and so on must be declared in normal Ada declarative parts.

2. A subprogram definition with type declarations for its parameters must be provided for every calling form.

3. Occurrences of the same calling form must be consistent with respect to the Ada types of arguments. If they are not, then the calling form is an overloaded subprogram call, and corresponding multiple definitions of the subprogram must be given.

With an Ada compiler of full capacity we can get more efficiency by the use of the INLINE pragma which causes the subprogram text to be substituted at the point of call. Conditional compilation can also be achieved by the use of conditional statements which the compiler executes at compile time. These features together constitute a primitive macro capability. They do not, however, increase the expressive power of the language; they only serve to make the compiled program more efficient, and possibly somewhat larger.

Language Extension

Procedures and functions are simple methods of language extension, but there is much more that we can do both in declarative and executable parts of a program. There are two levels of extensibility which can be found in contemporary extensible programming languages.

The first level of language extension is the ability to create abstract data types and operations on them. In some languages the programmer can create new operators for use in expressions, associating each operator with a specified function which takes one or two arguments. Ada provides this capability within limits. Only the predefined operators of the language may be used -- these become overloaded so that the function actually invoked depends on the operand types.

Another language feature with bearing on extensibility is the ability to hide the details of implementation of an abstract data type. In Ada the details of a data type may be hidden so that only the software implementing the defined operations can depend on those details. In Ada an abstract data type may be endowed with the language defaults for assignment and equality-test. As a further option, even these operators may be forbidden.

The second and more general level of language extension gives the user "self-replacing" calls as a counterpart to "value-returning" calls. As with subprograms, this kind of programming is a two-stage process of expression. The user writes a program in terms of self-replacing calls. He is then required to define an exchange rule, or "macro definition," which produces a program fragment to be substituted at the point of invocation.

This second level of language extension can be generalized by permitting the user to extend the syntax of the language when invoking these new constructs. Language extensions of this sort are called syntax macros [Leavenworth 66]. The alternative, as found in LISP and many macro assemblers, is to impose a uniform syntax on all macro invocations. Our present proposal uses the fixed syntax of Ada calling forms with limited syntactic entities in the argument slots.

The method of language extension by macro expansion can be used in support of Rapid Prototype Programming as a more powerful method of interpreting and expanding calling forms. This gives the user more control over the generated program since it gives the user language extension in declarations, in control structures, and in naming variables or data objects.

Programming Worlds

Having generalized the syntax of subprogram calls and having described a language extension mechanism, greater increases in power can best be achieved by providing families of functions in selected application areas. Certain high-level data types and operations have been found very useful and powerful in contemporary very high level languages. These include mathematical sets, sequences, functional mappings, pattern matching, and variants of

predicate calculus quantifiers as programming operators. Some languages of interest in this regard are SETL [Dewar 79], VERS2 [Earley 74], Madcap VI [Wells 72], and MLISP [Meehan 80]. Some examples of this kind of packaging can be found in Appendix A, but it is beyond the scope of this thesis to explore any of these programming domains in great detail.

Program Transformations

Program transformations are used by optimizing compilers for improving mechanically generated programs. A transformation system might be very valuable for improving the quality of code provided by in-line procedure substitution or by language extension features. Since the problem of improving program efficiency by program transformations has been studied elsewhere, we do not pursue this topic further in this dissertation [Standish 76, Loveman 77, Kibler 78, Smith 79].

We note that pattern-directed program transformations can also be used as a limited language extension capability. In this approach, pattern-directed substitution rules are given to provide meaning to novel language constructs. A possible extension to Castor would be to provide pattern-directed transformations on calling forms, based on the identifiers in the calling form name. This feature for building general purpose application packages is also beyond

the scope of this dissertation.

THE USE OF PARAMETERS

The Eight Queens program given in Chapter 4 shows the use of calling forms in a fairly basic way. In the refinements shown all data manipulations in the program are performed by parameterless procedures. Since there is no explicit indication of which procedures change (or access) which variables, all variables must be considered global.

In prototyping, a programmer may wish to write calling forms with parameters. This makes for a more detailed program but also helps limit the scope of the individual subproblems. Parameter notation is, of course, perfectly suited to situations in which the same operation is to be performed on different operands.

For an example of program development we consider the following program which plays Tic Tac Toe interactively against a human player. We omit the definition of "Board" in line 4 for the moment -- note, however, that we have allowed "Board(m)" to appear on the left of an assignment in lines 20 and 23.

```
1)  procedure Tic Tac Toe is
2)     type Move is
3)        record
4)           R,C: Integer range 1..3;
5)        end record;
6)     type Occupancy is (' ', 'X', 'O');
7)     Board: <>;
8)     Computer_Token,User_Token:
9)        Occupancy range 'X'..'O';
10)    Users_Turn: Boolean;
11)    Tokens_On_Board: Integer range 0..9  := 0;
12)    M: Move;
13) begin
14)    User_Token := Request User Choice Of Token;
15)    Users_Turn := (User_Token = 'X');
16)    if Users_Turn then Computer_Token := 'O';
17)    else Computer_Token := 'X';
18)    end if;
19)    loop
20)       if Users_Turn then
21)          Display (Board);
22)          M := Request A Valid Move On (Board);
23)          Board(M) := User_Token;
24)       else
25)          M := Generate A Move On (Board);
26)          Board(M) := Computer_Token;
27)       end if;
28)       Tokens_On_Board := Tokens_On_Board + 1;
29)       exit when (Tokens_On_Board = 9)
30)          or Game Is Won On (Board) By (M)
31)          or Game On (Board) Cannot Be Won;
32)       Users_Turn := not Users_Turn;
33)    end loop;
34)    Display (Board);
35)    if Game Is Won On (Board) By (M) then
36)       if Users_Turn then Humbly Congratulate;
37)       else Gloat Insufferably;
38)       end if;
39)    else Announce A Draw;
40)    end if;
41) end Tic Tac Toe;
```

This program was written virtually as is by a programmer (myself) who had never before written an algorithm to play this game. It captures all of the behavior and structure of the program while leaving the

details of strategy and winner detection unspecified.

A list of unspecified items provided by the Aide would be as follows:

```
type of Board
function Request User Choice Of Token
procedure Display (Board)
function Request A Valid Move On (Board)
function Generate A Move On (Board)
function Game Is Won On (Board) By (M)
function Game On (Board) Cannot Be Won
procedure Humbly Congratulate
procedure Gloat Insufferably
procedure Announce A Draw
```

SELF-REPLACING FORMS

So far we have used calling forms as procedure calls and function calls. In addition there are features we would like to have in a Rapid Prototyping language which cannot be realized by procedure and function calls, for example:

1. Data types (in type definitions or variable declarations)
2. Variable locator expressions (target of an assignment)
3. New control structures

As a first example, in the Tic Tac Toe program given above we would like to define Board as a map from the squares on the board (the space of possible Move's) to the contents of that square (an 'X', an 'O', or a blank). We write this (line 7) as follows:

7) Board: Map (Move) Into (Occupancy) Initially(' ');

This means that Board is to be a total function defined on the domain of possible Move's. The initial value for each move is ' '.

Note that the definition above is different from:

X: <u>array</u> (1..3, 1..3) <u>of</u> occupancy;

This latter definition could be used as a representation for the former, but they are logically different from the programmer's point of view -- the array has two arguments, while the map has one argument.

"Map (-) Into (-) Initially (-)" is a calling form which must be a <u>self replacing</u> call. This means that there is a definition which takes the calling form arguments as written and calculates a program fragment which is to replace the calling form in the text. In an environment where compilation takes place, this is to be done at compile time. In an interpretive environment, this replacement can be done the first time the calling form is encountered during interpretation. This kind of macro facility is also used in LISP [Meehan 79]. One way to handle "Map (-,-,-) Into (-) Initially (-)," for example, is to replace it with a program fragment of the form "<u>array</u> (-,-,-) <u>of</u> (-) := (-)" when the argument types are all discrete. Clearly there are many alternatives [Dewar 79].

Associated with this in-place replacement of the declaration must be some rules for interpreting references to the defined object in the subsequent text -- these are themselves other calling forms. For example, line 23 has the following statement:

```
23)    Board(M) := User_Token;
```

in which we must assign some meaning to the calling form "Board(M)." We might consider implementing this as a subprogram (function) call, but the only way to do this would be to have a function returning a pointer value. Even so, to achieve legal Ada we would have to rewrite the statement using ."all" as follows:

```
Function_name(M).all := User_Token;
```

Thus we have no choice: this calling form has to be rewritten by some macro definition. This is our second example of the need for self-replacing calling forms.

With the array implementation of Board described above, each occurrence of "Board(M)" in the program body can be rewritten as "Board_Variable(M.R,M.C)." This representation is equally valid both on the left and right of an assignment, given that the text is substituted in place.

To give a third example of self-rewriting forms we consider a definition that might be provided for the

procedure "Generate A Move On (Board)." Let us suppose the writer wants to use a plan notation of the following form (which does not conform to our notation of calling forms):

```
TRY stmt_1;
TRY stmt_2;
. . .
TRY stmt_n;
OTHERWISE stmt_n+1;
```

meaning

```
First try stmt_1.
If that doesn't work, try stmt_2.
. . .
If that doesn't work, try stmt_n.
If that doesn't work, then do stmt_n+1.
```

This control structure bears some resemblance to the exception handling facility of Ada, where a procedure or function can return normally or transfer control to an exception handler. A natural indication of failure of any "stmt_i" is to raise an exception during its execution. This leads to the following realization of the above program fragment:

```
begin
   stmt_1;
   goto done;
exception
   when others => null;
end;

begin
   stmt_2;
   goto done;
exception
   when others => null;
end;

. . .

begin
   stmt_n;
   goto done;
exception
   when others => null;
end;

stmt_n+1;

<<done>>
   null;
```

In this realization, the statements to be attempted have been embedded in a sequence of begin blocks, each of the form:

```
begin
   stmt_i;
   goto done;
exception
   when others => null;
end;
```

This means that "stmt_i" is executed, and in the normal case control passes to the next statement. This in turn is a goto statement which transfers control to the end of the

sequence. If the statement generates an exception, then control is given to the exception handler at the end of the begin block. The Ada notation here means that when any exception is encountered a null statement is executed and then control passes out of the begin block and on to the next block in sequence.

We consider now how to represent our plan in self-replacing calling forms. A straightforward description of the plan would be as follows:

```
Try (Win On (Board) For (Computer_Token));
Try (Block (Win on (Board) For (User_Token)));
Try (Fork On (Board) For (Computer_Token));
Try (Block (Fork On (Board) For (User_Token)));
Otherwise (Random Move On (Board));
```

The problem with this notation is that "Try" and "Otherwise" cannot be defined using our macro expansion paradigm. This is because they will perform substitutions only at a level below the sequencing implied by the semicolons. The resulting substituted statements would all be executed, regardless of the success of any preceding attempts. Therefore these statements must be combined into a single calling form invocation, as shown in the following refinement of "Generate A Move On (Board)":

```
1)  function Generate A Move On (B)
2)     return Move is
3)  begin
4)     Attempt Plan
5)        begin
6)           Win On (B) For (Computer_Token);
7)           Block (Win On (B) For (User_Token));
8)           Fork On (B) For (Computer_Token);
9)           Block (Fork On (B) For (User_Token));
10)          Otherwise (Random Move On (b));
11)       end;
12) end Generate A Move;
```

```
1)  function Block (M) return Move is
2)  begin
3)     return M;
4)  end;
```

We will consider the mechanics of expanding self-replacing calling forms in the next section.

CALLING FORM MACRO EXPANSIONS

There are two settings in which macro expansion may take place: during interpretive execution of a program or during static refinement of a Castor program to pure Ada text. In either case the same macro definition is applied, although the operations that it directs are implemented differently. In Castor only dynamic expansion is provided. Macro calls are invoked as they are encountered, outermost first. We use the term *macro expansion* or *macro execution* to refer to the process of generating a program fragment to be substituted at the point of invocation.

The language in which macros are written is the same language that programs are written in. Special data types and operators are provided for performing the actual program modifications. In this way the specific details of the internal program representation are hidden from the user.

Macro Definitions

We classify macros as subprograms in Castor, although they are quite different in meaning from procedures and functions. As with procedures and functions, a definition must be provided for each macro that is used. In keeping with the pattern of Ada, a "macro specification" defines the parameters and results of a macro, and a "macro body" gives a macro specification followed by all internal information. Since, as we shall see, a macro has only one kind of argument, there is no need for generic macros. A typical macro specification might be as follows:

```
macro A (X) B (Y1,Y2) C (Z) return T;
```

while the corresponding macro body would be written in the following form:

```
macro A (X) B (Y1,Y2) C (Z) return T is
   -- declarations go here
begin
   -- statements go here
end;
```

A macro is like a function in that the type of the result is

declared and a *return* statement must be executed to identify the returned value and end the macro execution.

Data Types Used In Macros

No types are specified for macro parameters because all macro parameters are of the same type, "Intnode." This type definition is provided as part of the predefined package MACRO_STANDARD. All identifiers in this package are automatically visible in any macro body, just as identifiers from the package STANDARD are initially visible in all programs. This package also provides a number of functions for manipulating internal program structures.

The internal form of a program is assumed to be a parse tree of the program according to a grammar resembling the Reference Grammar of Ada [Ada 80]. The details of the grammar used are not specified for our purposes, nor is explicit traversal of the tree necessary.

The type of any node of the tree is "Intnode." This type is *private*, meaning that the only operations that can be applied are assignment, equality test, and those provided in the MACRO_STANDARD package. We specify five additional types, corresponding to the kinds of node we can generate with a macro. These are as follows: a type, a declaration, a name, an expression, and a statement. The following definitions appear in the visible part of the MACRO_STANDARD package:

```
type Intnode is private;
type Typenode is private;
type Declnode is private;
type Namenode is private;
type Exprnode is private;
type Stmtnode is private;
```

The operations provided by MACRO_STANDARD are suitably overloaded so that any of these five types can be converted to an Intnode and used as such. Intnodes can be moved around and put in lists, but to be converted to one of these five node types (and hence to affect the result of the macro) an Intnode must be unparsed and reparsed by a function which generates one of these five designated types of node. This guarantees the syntactic Validity of any of these nodes as the corresponding kind of phrase.

In addition to the above, we have the following:

```
type Intnode_List is private;
type Typenode_List is private;
type Declnode_List is private;
type Namenode_List is private;
type Exprnode_List is private;
type Stmtnode_List is private;
```

These are data structures which represent lists of the various node types described above. In addition to the types mentioned above, a macro in the operating version of Castor may also return a Stmtnode_List or a Declnode_list. This could meaningfully be extended to include names and expressions as well.

Syntactic Validation Of Macro Expansions

We wish to ensure that the program after macro expansion is syntactically valid -- that is, that the internal program representation, after macro expansion, represents the parse tree of a syntactically valid program. We do this by defining five different nonterminals in the syntax of Castor:

```
type_calling_form
decl_calling_form
name_calling_form
expr_calling_form
stmt_calling_form
```

For simplicity each of these nonterminals is defined by a production with the same right-hand side. However, each of these nonterminals can only appear in the right-hand side of one rule. For example, "stmt_calling_form" can only occur in Castor as a statement. A calling form appearing as an expression, on the other hand, will only be recognized as an instance of "expr_calling_form." The appropriate calling form nonterminal must be recorded at the corresponding node of the parse tree.

To check the syntactic validity of a macro expansion, the interpreter proceeds as follows:

1. It encounters a calling form node which has the name of a macro.

2. It determines the nonterminal associated with this node -- say, "type_calling_form."

3. It checks this nonterminal against the return type declared in the macro definition -- in this case it must be the subtype "Typenode."

4. It executes the macro and replaces the calling form node with the returned program fragment. In the case of "Declnode_List" or "Stmtnode_List," a list is returned which is spliced into the list of corresponding items in the enclosing program.

In order to guarantee the syntactic validity of the resulting program, it is sufficient to guarantee that the data item computed within the macro is indeed a program phrase of the corresponding type. But this is just the property we have stipulated above.

This syntactic checking can be performed efficiently and involves only local information from the parse tree and information about the macro which may be stored in a dictionary. We note that this checking described so far can be done statically, if desired, without expanding any macro calls.

Generating Program Fragments

The operators which synthesize or modify program fragments are also provided in the MACRO_STANDARD package. For portability and user convenience they are oriented toward the surface representation of Ada programs. The following five functions take a character string, parse it, and return a node of the appropriate type. Associated with each function is an implied nonterminal which is the "start

symbol" for the parse.

```
function Gentype(S:String) return Typenode;
function Gendecl(S:String) return Declnode;
function Genname(S:String) return Namenode;
function Genexpr(S:String) return Exprnode;
function Genstmt(S:String) return Stmtnode;
```

The following are examples of valid calls:

```
T := Gentype("Integer range 5..10");
D := Gendecl("X: Boolean;");
```

To improve readability, we allow an ending semicolon to be omitted:

```
S := Genstmt("exit when B;");   -- is the same as
S := Genstmt("exit when B");    -- (no semicolon)
```

Substituting Into Program Skeletons

In general a macro must perform substitution into skeleton program fragments. The string argument S has a special form when substitutions are desired. Each point of substitution is denoted by a dollar sign preceding an identifier, for example "$A." (Note that dollar sign is an otherwise illegal character.) The identifier is the name of a variable or parameter in the macro environment and must be an Intnode or one of the other node types. The current value of this variable is unparsed and is substituted into the string before parsing begins. If a particular item is to be substituted into two places in the string, it is

simply used twice in the list. Note that these functions are by no means ordinary Ada functions since they have access to variables in the macro in this unusual manner.

The following is an example of substitution:

```
S1 := Genstmt("X := X+Z");
S2 := Genstmt("exit when X>100");
S := Genstmt("loop $S1; $S2; end loop");
```

This is equivalent to

```
S := Genstmt (
   "loop "           &
      "X := X+Z; "     &
      "exit when X>100; "    &
   "end loop");
```

(Ampersand, "&," denotes concatenation, permitting a string to be broken over consecutive lines.) The items substituted by this means may be constructed by previous computations, like S1 and S2 above, or may be formal parameters of the macro.

This scheme for manipulating program fragments has the advantages that it is straightforward to implement, it is easy for the macro programmer to get used to and remember, and the skeleton program fragments are visually similar to the source language. The user has no need to know or remember the actual internal representations of program phrases.

Aggregates of Nodes

Frequently we wish to deal with groups of nodes, often in varying numbers. We therefore provide macros that can accept a variable number of arguments in a given position. This is indicated in the macro specification by the keyword *array* as an argument type, for example:

```
macro The Set (V: array) return Exprnode;
macro Loop Forever (Z: array) return Stmtnode_List;
```

The formal parameter so indicated is of type "Intnode_List." This is not really an array, but rather a list; "*array*" is used because it is a reserved word that is suggestive of the true meaning.

Parameters are furnished in the invocation of such a macro simply by giving as many as desired in the appropriate place, for example:

```
X := The Set();
X := The Set(A);
X := The Set(A,B,C);
Loop Forever
   begin
      Copy Input To Output;
      Report Progress;
   end;
```

In order to manipulate lists of nodes the following set of representative functions is defined:

There is a fine point to consider when splicing a list of nonterminals into a program fragment. It is not sufficient simply to unparse the list of nodes and substitute the result into the program string before parsing. The parser expects items in lists to be separated by either semicolons or commas, depending on the context. In order to specify whether commas or semicolons are to be provided, the substitution mechanism checks to see if a semicolon immediately follows the substitution locus in the target string. If not, commas are placed between the items. If a semicolon is found, then a semicolon will follow each item, unless the number of items is zero, in which case the flag semicolon will be removed. Castor supports just these two delimiters, though experience may indicate whether other kinds of list delimiter tokens are also desirable.

Additional Features

In addition to the above capabilities, there are some miscellaneous functions which are required in practical macros. The following subprograms have been found to be useful:

```
function New_Id (I:Intnode;S:String) return Namenode;
function New_Id (S:String;I:Intnode) return Namenode;
function New_Id (I1,I2:Intnode) return Namenode;
function Test_Decl (I:Intnode) return Boolean;
procedure Add_Decl (D:Declnode);
procedure Set_Tag (I:Intnode; K:Integer; V:Intnode);
function Tag (I:Intnode; K:Integer) return Intnode;
function Type_Of (E:Intnode) return Namenode;
function Def_Of (I:Intnode) return Intnode;
procedure Match (I:Intnode; S:String);
function Subst(New,Old,Body:Intnode) return Intnode;
```

Note that where a parameter of type Intnode is specified an actual parameter of any of our restricted node types may also be furnished.

The function New_Id is used to create a node representing a new identifier. The Intnode argument or arguments each represent an identifier or a series of identifiers (as in a calling form). The character string (if present) is also an identifier. These identifiers are concatenated with the underscore character, "_," as a separator, and this new identifier is the result.

The function Test_Decl takes a node representing an identifier and returns the value True if that identifier is as yet undefined.

The procedure Add_Decl adds the given declaration to the innermost enclosing declarative part.

The procedure Set_Tag is used for making arbitrary extensions to the symbol table. The first parameter is an identifier and the second is an arbitrary integer. This integer selects a value slot in the symbol table extension

for the identifier. No such slots are predefined in meaning. The third parameter is an arbitrary Intnode to be placed in the slot.

The function Tag is for accessing a value stored by Set_Tag. The first parameter is the specified identifier, and the second parameter is the slot number.

The function Type_Of takes an Intnode representing an expression, variable, or type identifier and determines its base type. The identifier associated with the base type is returned. If the argument is a macro calling form, it is expanded before the above determination.

The function Def_Of takes a type identifier as its argument and returns an Intnode representing the type definition of that identifier. This is the program fragment previously specified in a statement of the form:

    type identifier is type_definition;

In this definition all record fields, "A,B,C: T," are distributed, "A:T; B:T; C:T;," and all subtype indications for discrete types are expanded to have the form

    Typeid range Minvalue .. Maxvalue

The procedure Match takes an Intnode and matches it against a program phrase in a character string. Pattern variables may be indicated in the character string by use of

the dollar sign, just as for the parsing functions. "$A" indicates that "A" is to be assigned the corresponding sub-phrase; "$$A" indicates that a list of sub-phrases is to be matched. If no match is found, an exception is raised.

The function Subst is used to substitute an Intnode, "New," for all occurrences of a second Intnode, "Old," which occur in a third Intnode, "Body." This modified copy of Body is returned.

We have shown in some detail a powerful but straightforward method for building program fragments in an implementation independent way. These features are sufficient for writing practical macro expansion definitions, but future experience will undoubtedly reveal other desirable features. Examples of the use of Castor macro definitions may be found later in this chapter and in Appendix A.

PROGRAMMING WORLDS

At this point we suppose the programmer wishes to write the detailed algorithms for playing Tic Tac Toe. These are most naturally expressed in a mathematical notation in the style of SETL [Dewar 78] or VERS2 [Earley 74]. We shall see that high-level operators of this kind can also be expressed quite conveniently and legibly in Ada calling forms.

One of the first things we want to do is define the set of paths in the Tic Tac Toe board. To paraphrase the following mathematical statement

Let Row(I) = {Move'(I,C) | C = 1..3}

we could write the following calling form:

```
(Row(I)) IS
   (The Set(Move'(I,C)) For(C) In(1..3));
```

(The notation Move'(I,C) is an Ada "qualified expression" and means that an object of type Move is to be constructed from I and C.)

Considering how we might wish to implement this, some alternative possibilities come to mind. First, we might wish "Row(I)" to be a function:

```
function Row(I:Integer range 1..3)
   return Set Of(Move) is
begin
   return The Set(Move'(I,C)) For(C) In(1..3);
end;
```

This, however, would require some fairly deep analysis to discover a type for "I" and for the return value. We could force the programmer to provide this additional information, but there is an easier way.

Another possibility would be to define a map and initialize its values accordingly. Again we have the problem of discovering a domain and a range type.

The simplest way to interpret this calling form is to use the textual substitution model for subroutine invocation. We simply replace every occurrence of

```
Row(X)
```

with

```
The Set(Move'(X,C)) For(C) In(1..3)
```

This eliminates the problem of finding the type of I and of the return value. There is a potential for conflict with the bound variable C, but it is natural to expect the calling form "The Set(-) For(-) In(-)" to bind this variable in a _begin_ block.

To accomplish this substitution we use the function Subst(A,B,C), introduced above, which substitutes Intnode "A" for all occurrences of Intnode "B" in Intnode "C." We want the following macro definition to be created:

```
macro Row(Param) return Exprnode is
   Free_var: Intnode := Tag(Genname("Row"),1);
   Expr    : Intnode := Tag(Genname("Row"),1);
begin
   return Subst(Param,Free_var,Expr);
end;
```

To do this, "Is" must be defined as follows:

```
macro (Lhs) Is (Rhs) return Declnode is
   Name,Free_var: Intnode;
begin
   Match(Lhs, "$Name($Free_var)");
   Set_Tag(Name,1,Free_var);
   Set_Tag(Name,1,Rhs);
   return Gendecl (
      "macro $Name(Param) return Exprnode is            "&
      "Free_var: Intnode :=Tag(Genname(%$Name%),1);"&
      "Expr     : Intnode :=Tag(Genname(%$Name%),2);"&
      "begin                                             "&
      "   return Subst(Param,Free_var,Expr);            "&
      "end; ");
end;
```

(The percent character is an alternative string delimiter in Ada.) Introduction of this capability gives us great flexibility in using mathematical calling forms, as we shall see.

A similar calling form, "(-) If (-)," can be used to perform the identical function described above for "Is." This reads more meaningfully when the map or function is boolean valued (a predicate). For example,

```
((X) Is Even) If (X mod 2 = 0);
```

defines the predicate "(X) Is Even" in the natural way.

The calling form

```
The Set (F(X)) For (X) In (S)
```

is similar to the "iterators" of SETL and VERS2. It might be rendered in English as "the set of all Y such that Y=F(X) for some X in S," or "the set formed by taking each X in S

and collecting the values F(X)." This is implemented as a macro whose expansion depends on the particular representation chosen for sets.

We note the need and usefulness of expressions of this kind, without attempting to define a comprehensive repertoire. This kind of package of definitions may become standardized in years to come. Some additional calling forms are as follows:

```
The Set (A)            the singleton set {A}

The Set (A,B)          the set {A,B}

The Set (X) In (S) St (P(X))
                       the set of all X in S such that
                       P(X) is true

The Set (F(X)) For (X) In (S) St (P(X))
                       the set of all values F(X) where X
                       is in S and P(X) is true
```

Similar iterative operations based on predicate calculus and having parallels in other languages are:

```
Exists (X) In (S) St (P(X))
                      a boolean valued calling form which
                      returns the value True and sets the
                      value of X when a suitable X can be
                      found in S

Any (X) In (S) St (P(X))
                      a function which sets the value of
                      X if such an X is found and which
                      also returns the value of X --
                      otherwise an exception is raised

Find (X) In (S) St (P(X));
                      a statement similar to the above --
                      X is set if such a value is found,
                      otherwise an exception is raised
```

```
For (X) In (S) Assert (P(X))
For (X) In (S) St (Q(X)) Assert (P(X))
                            a boolean valued calling form which
                            returns True if P(X) is True for
                            all suitable values of X
```

These calling forms, though limited to sets for our purposes, provide a very flexible and natural means for rapid writing of both specifications and programs. Straightforward implementation of each of these is possible, and while the resulting implementation may be inefficient, this is of much less concern in a prototyping situation than in a production programming environment. This can be seen in the following very natural definitions for the Tic Tac Toe program:

```
1)  (Row(I)) Is (The Set(Move'(I,C)) For (C) In (1..3)));
2)  (Col(I)) Is (The Set(Move'(R,I)) For (R) In (1..3)));
3)  (Dia(1)) Is (The Set(Move'(I,I)) For (I) In (1..3)));
4)  (Dia(2)) Is
5)     (The Set(Move'(I,4-I)) For (I) In (1..3)));

6)  (The Rows) Is (The Set (Row(R)) For (R) In (1..3));
7)  (The Columns) Is (
8)     The Set (Col(C)) For (C) In (1..3));
9)  (The Diagonals) Is (
10)    The Set (Dia(I)) For (I) In (1..2));
11) (The Paths) Is (
12)    The Rows + The Columns + The Diagonals);
```

```
1)  (Game Is Won On (B) By (M)) If
2)     (Exists (P) In (The Paths) St (
3)        (M) In (P) and
4)        For (X) In (P - The Set (M))
5)           Assert (B(X) = B(M))));
```

expressions in a data aggregate. For splicing we use another special notation in the skeleton program, as we did above for substitutions. A locus for splicing is indicated by two dollar signs followed by an identifier, for example "$$A." This identifier denotes a macro variable or parameter which is a list of nodes. The following is an example of this kind of splicing:

```
A: Stmtnode_List;
B: Exprnode;
S,C: Stmtnode;

Clear List (A);
Appendl (Genstmt("X := X+Z")) To (A);
Appendl (Genstmt("Put(I)")) To (A);
B := Genexpr ("X<100");
C := Genstmt("I := I+1");
S := Genstmt("while $B loop $C; $$A; end loop");
```

The result is equivalent to the following:

```
S: Stmtnode;

S := Genstmt(
   "while X<100 loop "    &
      "I := I+1; "    &
      "X := X+Z; "    &
      "Put(I)  ; "    &
   "end loop");
```

Our use of the word "splicing" indicates that the nodes of the list are successively unparsed -- this process does not have to check any structure into which the items are being spliced, however. If there is a problem in this context it will be detected by the parser.

```
procedure Clear List (L: Intnode_List);
function Is Empty (L: Intnode_List) return Boolean;
procedure Push (X: Intnode) On (L: Intnode_List);
function Pop (L: Intnode_List) return Intnode;
procedure Appendl (X: Intnode) To (L: Intnode_List);
```

"Clear List" makes a list empty. "Is Empty" is a predicate to test whether a list is empty. "Push (-) On (-)" adds an element to the beginning of a list, and "Pop" removes an element from the beginning of a list. "Appendl (-) To (-)" adds an element to the end of a list.

As we mentioned above, it is necessary to overload these operations so that conversions from our special node types to Intnode are permitted and the reverse conversions are not. This is achieved by the following:

```
procedure Clear List (L: Typenode_List);
function Is Empty (L: Typenode_List) return Boolean;
procedure Push (X: Typenode) On (L: Typenode_List);
procedure Push (X: Typenode) On (L: Intnode_List);
function Pop (L: Typenode_List) return Typenode;
function Pop (L: Typenode_List) return Intnode;
procedure Appendl (X: Typenode) To (L:Typenode_List);
procedure Appendl (X: Typenode) To (L:Intnode_List);
```

Corresponding definitions are also given for Declnode's, Namenode's, Exprnode's, and Stmtnode's.

It is also necessary to be able to splice a list of internal nodes (for example aggregate parameters like V and Z above) into a program skeleton to form a list or part of a list of items in the language. For example, we may have a sequence of statements in a loop body or a sequence of

```
1)   (Win On (B) For (Token)) Is
2)      (Find (M) In (Move) St (
3)         Exists (P) In (The Paths) St (
4)            (M) In (P) and
5)            B(M) = ' ' and
6)            For (X) In (P - The Set (M))
7)               Assert (B(X) = Token))));

1)   (Fork On (B) For (Token)) Is
2)      (Find (M) In (Move) St (
3)         B(M) = ' ' and
4)         Exists (P1,P2) In (The Paths) St (
5)            For (P) In (The Set (P1,P2)) Assert (
6)               (M) In (P) and
7)               Exists (M1,M2) In (P - The Set(M)) St (
8)                  B(M1) = ' ' and
9)                  B(M2) = Token)))));
```

CHAPTER 6

INTERACTIVE PROGRAM MANIPULATIONS

Various models of software development have been advanced which view the design as a step by step process. Such models include "top-down" design, "bottom-up" design, "module-by-module" design, and others [Freeman 80]. No realistic model, however, proposes that design can proceed monotonically by any known method without iteration and backtracking. The essence of program development is the process of feedback, incremental evolution, and convergence to a solution. One of the features of the Aide is to offer program transformations which arise frequently during program development. Of particular interest are transformations which are conceptually simple but lengthy to describe in basic terms.

Given support for the mechanics of program change, it is still best if changes can be avoided altogether. We can do this by reducing the information that must be specified and by reducing the redundant information distributed through a program. In short, we wish to take liberties with the rules of the language in order to reduce the verbosity

of programs. This not only makes the initial articulation of the program easier, but also means there is less to change when changes become necessary. Localizing the information relevant to individual design decisions makes it easier to remake those decisions and update the program representation. Also the less redundant, distributed information there is, the less thought and effort are required in the first place to make the program complete and consistent with itself.

In general our approach is to provide rules and mechanical means for transforming such programs into pure Ada so that standard compilers and other tools can be used. This set of features in the Aide we call a "laundry," since it takes sloppy or "dirty" programs and cleans them up.

In this chapter we first consider features of the Ada "laundry" and transformations involved in its operation. We also discuss some high-level program editing transformations and conclude with some suggestions for interactive programming.

EXPLOITING DATA TYPES

The above considerations should not be construed as an argument for terse, cryptic language features. Certain kinds of redundancy are conceptually helpful and conducive to orderly thinking and correct problem solving. One of

these is type declaration and type checking as found in languages like Ada and Pascal. The information provided permits both static and run-time checking of the program for consistency. For example, the user can be protected against passing an integer as a floating point parameter, exceeding the bounds of an array, or using the wrong template for the data area referenced by a pointer. We consider first some ways to streamline the data typing facilities in Castor.

Reducing Verbosity of Data Types

It is time consuming and wordy to supply the type definitions of procedure or function parameters. If we are doing top-down programming and have already written one or more calls to a procedure, the types of the formal parameters can be inferred from the types of the actual parameters in the procedure call or calls. We can put this burden of type attachment on the Aide -- in general these induced types should be taken to be unconstrained. Note that consistency checking may be done among several actual calls, and the user may be informed that overloaded definitions are required.

Another feature of Ada is that complex types must be defined by chains of definitions, with type identifiers describing the intermediate constituent types. This makes for a certain style in Ada programming and forms a special kind of self documentation in the program. From the point

of view of flexible program changes this is quite awkward, however. The intermediate identifiers and their declarations are burdensome and redundant, and if major changes are made in the program, the investment in writing them may be wasted. It is much handier to employ special purpose editing transformations which will generate the appropriate correct Ada representation. Suppose that the following type declaration is input (in the style of Pascal, lacking intermediate type identifiers):

```
author:
   record
      name:
         record
            first, middle, last: string;
         end record;
      residences: access array (1,2) of city_address;
   end record;
```

The Ada laundry automatically introduces the required intermediate type identifiers. As an option the user may wish to choose the specific identifiers to be introduced, as in the following:

```
name_type:
   record
      first, middle, last: string;
   end record;
residence-list: array (1,2) of city_address;
residence_list_ptr: access residence_list;
personal_data:
   record
      name: name_type;
      residences: residence_list_ptr;
   end record;
author: personal_data;
```

In this example there are four type identifiers the programmer did not have to create, check for uniqueness, and specify in his first writing of the program.

There are two ways in which we may wish to display a Castor program. We may wish to view it and edit it as written, or we may wish to see the _inferred_ parts of the program displayed as if they had been written by the user. Thus it may be desirable to have a print-out which distinguishes "real" text from "inferred" text -- as in editions of the King James Bible where words are italicized which are interpolations into the text of the original language. This is done by implementing these transformations as automatic program refinements which introduce program attachments, as described in Chapter 4. In this way the user retains the ability to make localized changes and have the Aide automatically update the "inferred" areas of the program.

Type Checking

We offer the user a chance to specify data types without having to create intermediate type identifiers, as we have said. As a programming option the user may wish to adopt a style where he avoids type identifiers and simply specifies type structures. For example,

```
X: array (1..5) of integer;
Y: array (1..5) of integer;
```

In strict Ada, these variables X and Y have different types and are therefore incompatible for assignment, passing as parameters, and so on. The user may wish to loosen the type checking rules of Ada to make such variables compatible, as in Pascal.

Uniform Notation

Ada has a variety of notations which can all represent the following abstract relationship:

The (A) Of (B)

They are,

B(A)	Element (A) Of Array (B)
B.A	Field (A) Of Record (B)
A(B)	Function (A) Applied To (B)
B'A	Attribute (A) Of (B)

This is a somewhat confusing set of alternatives. For example, if "Fred" is a data item denoting an individual, the age of "Fred" might be represented and accessed in several different ways:

Fred(age)	"Fred" is an array, "age" is an index value
Fred.age	"Fred" is a record containing an "age" field
age(Fred)	The age is computed by function "age"

(An attribute cannot be used in this example because in Ada all attributes are predefined and are fixed in meaning.) If we decide to change from the record representation to the

functional representation, for example, we must change the appropriate declarations and must change all references of the form "Fred.age" to "age(Fred)." This is also necessary for all other objects with the same type as "Fred," and may also be necessary for other operators besides "age."

The least that the Aide can do is provide assistance in finding and changing all of the appropriate references in a situation like this. The following discussion provides motivation for additional assistance from the Aide.

In three of these forms the "operator," A, is written to the right of the "operand," B, and in one form A and B are written in the opposite order. This varied left-application and right-application can be an impediment in writing and inspecting a program. When writing a nested expression it is much easier to follow the sequence of operations if all the operators are left binding (or all are right binding). For example, suppose that T1, T2, T3, T4, and T5 are data types and we have the following declarations:

```
function A(p: T2) return T1;
function B(p: T3) return T2;
function C(p: T4) return T3;
function D(p: T5) return T4;
X: T5;
```

The following is quite easy to program and easy to read, either from left to right or from right to left:

```
A(B(C(D(X))))                                    6-1)
```

D, C, B, and A are applied in order and it is easy to do type checking visually. D is well defined on X (type T5), C is well defined on any output of D, B is well defined on any output of C, and A is well defined on any output of B and returns a value of type T1.

We now consider what happens if we decide that D and B can be "efficiently" represented as fields within record types. The equivalent definitions are as follows:

```
function AA(p: TT2) return TT1;

type TT3 is
   record
      . . .
      BB: TT2;
      . . .
   end record;

function CC(p: TT4) return TT3;

type TT5 is
   record
      . . .
      DD: TT4;
      . . .
   end record;

XX: TT5;
```

Our expression now becomes:

```
AA(CC(XX.DD).BB)                                 6-2)
```

We can check that DD is well defined on XX, CC is well defined on XX.DD, BB is well defined on CC(XX.DD), and AA is

well defined on CC(XX.DD).BB and returns a value of type TT1. It is more difficult to check this, however, since the definitions for AA, BB, CC, and DD are now dissimilar, and we have to scan the expression from the "inside" outward, checking which operator is applied at each point.

Visual inspection of expression 6-2 does not readily show what is going on -- namely, that we are taking a single value, XX, and are applying four operators to it in sequence. Nor is the sequence in which the operators are applied clearly shown. If we read from left to right, we find AA applied and CC applied, but as we go on we are surprised to find BB applied in between! Expression 6-2 cannot be read from right to left either. The eye must somehow find the "center" of the expression and then work from the inside outward, as we did above in doing the type checking. Naturally, this becomes even more complicated when the functions take more arguments and the other arguments are also compound expressions. The programmer who writes these expressions has similar difficulties.

One might argue that the fault in the preceding predicament is in the programmer who writes expressions which are too complex. We could easily rewrite expression 6-2 as follows:

```
XX1 := XX.DD;                       6-3)
XX2 := CC(XX1);
XX3 := XX2.BB;
XX4 := AA(XX3);
```

and then use XX4 in the context where expression 6-2 was to be used. This is counter-productive for Rapid Prototyping, however, because it slows the programmer down, returning to a style of programming resembling assembly language. It also requires the declaration of otherwise useless variables and the creation of unique names for them.

The problem with 6-2 lies in the inconsistent association of the various functional notations -- sometimes requiring an operator to be written on the right and sometimes on the left. The problem arose in this case because of the user trying to take advantage of an "efficient" record implementation too soon. If the programmer had initially written everything with left-applied functions, all would be well. But it is all too easy to leap to the use of record types or some other "efficient" implementation just through the force of habit.

The answer we propose is to use left-applied operators at all times. This can be achieved by automatically defining suitable calling forms whenever a record type is introduced. This also relieves the problem of changing from one representation to another -- in the case of records and functions -- since the notation at points of reference remains unchanged.

Instead of attempting to work arrays into this scheme, we suggest that they should only be used for applications

which are clearly array-like in nature, and in case of any uncertainty the functional notation should be used.

INTERACTIVE PROGRAMMING CONSIDERATIONS

Automatic Command Completion

Given the heavy use of lengthy calling forms in a large program, it may become a troublesome clerical task to ensure that all references to the same calling form are spelled and worded exactly the same. Similarly the programmer may not remember the exact wording or choices available for a library calling form. In an interactive environment it is possible to lighten this burden by having the system help with typing in the calling forms. The TENEX or TOPS-20 operating systems provide a facility of this kind for automatic command completion. The user types in as much of a lengthy command as is necessary to identify it uniquely. Then instead of typing the remainder he may press the Escape key, and the system automatically fills in the rest of the command text. We can envision a similar facility for managing calling form names. If there remain parameters to be provided, the automatic type-in fills in up to the next left parenthesis and then waits for the user to complete the parameter with a right parenthesis, at which point automatic type-in can be continued using the Escape key. If the

automatic type-in reaches a point of ambiguity, the user can request a menu of choices with the question mark key. After disambiguation has been provided, type-in may be continued by the Escape key.

Deeply Nested Calling Forms

When calling forms are deeply nested, help may also be required in balancing parentheses. The use of square brackets ("[..]") as "super parentheses," as in LISP, is useful for this purpose [Meehan 79]. By convention, a right bracket matches as many left parentheses as necessary to go back to the preceding left bracket. If there is no matching left bracket, it matches as many left parentheses as necessary to close off the expression.

CHAPTER 7

RELATED WORK

Software Prototyping is a new topic in computer science, although, as we have noted, there is growing interest in this subject. A body of literature on Prototyping has yet to appear, but there are a number of related areas that have a bearing on our approach.

PROGRAM DESIGN LANGUAGES

In his paper on Stepwise Refinement, Wirth employed a Pascal-like language using descriptive procedure and function names as place holders [Wirth 71]. The term "stepwise refinement" was introduced in this paper as a concept for training programmers in systematic, top-down design. This high-level programming style was evidently conceived as a teaching tool and a manual tool students could use in developing their own programs. The Eight Queens example in Chapter 4 is taken from this paper.

Caine's Program Design Language, or "PDL," represents a similar use of a formal program structure enclosing English action descriptions [Caine 75]. A syntax is defined using a

small set of reserved words and structured statement formats. PDL is a documentation tool and is used to record high-level program design information in a machine readable form. This data base can be edited and updated as the design progresses and serves as system documentation and a guide for the program implementation phase. Various documents can be generated from the data base, including a table of contents, the text of the data base in pretty-printed form, a cross reference, and a listing of reference nestings. PDL was developed to produce human-readable documentation rather than executable programs, and so refinement of programs is by strictly informal means.

A number of other PDL's are in use, although as a rule these languages tend to be developed and used internally within commercial organizations rather than being described in the public literature. The PDL of Caine, Farber and Gordon, Inc., described above is a notable exception. A similar though unsupported PDL is described in [Zelkowitz 79]. Another PDL using the program structuring of Ada is under development along with a corresponding set of analysis tools at TRW [Hart 81]. This language is called "Ada PDL" and is characterized by the use of free format (uninterpreted) text for program statements, package declarations, and type definitions. IBM Federal Systems

Division has taken a more restrictive approach with their "PDL/Ada," using a proper subset of the syntax of Ada and allowing free format descriptions only in comments [Waugh 80].

As we have discussed earlier, the purpose of a PDL is to be a semi-formal design aid. Although some PDL's offer various kinds of automatic analysis, in general there is no mechanical assistance past the design stage. In fact, the PDL and the implementation language may be totally unrelated. A synchronization problem can therefore arise between the design data base and the executable code, since changes made during implementation (or later on) may or may not be properly incorporated back into the design data base. The approach of Castor, by contrast, is to provide a formal interpretation of the free form text used in high-level design. In this way the program is represented in only one medium instead of two, and there is continuity between the design and implementation phases of the development. The Harvard PDS system described below also addresses this problem.

ANNOTATION OF PROGRAMS

ANNA is a language being developed and studied at Stanford which is an extension of Ada to include formal annotations concerning program behavior [Krieg 80]. ANNA

uses formal comments of two kinds: virtual Ada text and annotations. These appear to an Ada parser to be comments, but they have their own particular syntax and semantics. These formal comments are intended for use in program verification and for writing formal program specifications. They resemble Castor refinements which are also displayed as comments. Furthermore, in both systems an attachment has an associated scope. In ANNA the scope of a declarative formal comment is like that of a declaration, and the scope of a statement formal comment is the preceding statement. Castor attachments have to be more general since refinements can operate on parts of statements or on multiple statements. Given certain extensions in the language to handle the text within attachments, an environment supporting Castor refinements could also readily support ANNA annotations.

INTERACTIVE DESIGN SYSTEMS

The PDS system (Program Development System) built by Cheatham _et al_ formally manages the levels of refinement of a program [Cheatham 79, Conrad 76]. This has been done in an extensible language environment in such a way that user-supplied refinement rules can be used to generate alternative implementations of the language extensions. In PDS a system data base is provided so that multiple representations of a program module are all available. A

high-level module can be modified to yield another version by interactive commands or by a saved series of commands. The transformations leading from one version of a module to another are remembered by the system as a history of the derivation, and in the event of program modifications the history can be used to replay the derivation sequence.

In the PDS system a program can be refined by manual editing or by applying a rewrite-rule facility. A refinement transformation of the latter sort is a pattern-directed program transformation. This rewrite facility can be used to supply the definition of a previously undefined program construct. It can also be used to improve executable code by replacing or restructuring a computation. Rewrite rules, like subroutines or macros, are viewed as a two-level method of writing a program, rather than as a means of communicating with a standard system library.

The Harvard PDS system and Castor are similar in philosophy. The PDS rewrite facility is like the Castor macro facility as a mechanism for refining undefined program fragments. While Castor macros are invoked by name, PDS macros are invoked by pattern matching on the source program. Macro invocation is therefore more efficient in Castor but more general in PDS, since the occurrence of consecutive expressions or statements can be used to trigger

a replacement. Expression of transformations is almost certainly easier in PDS when it happens that the desired replacement can be expressed as a pattern substitution. The designers of the PDS rewrite facility foresaw, however, the need to substitute computed program fragments. Castor handles this situation more uniformly and does so in a language that is largely independent of the internal program representation. PDS macros on the other hand, like LISP Macros [Meehan 79], must build the exact list structure to be substituted, piece by piece. The two systems also take a similar approach to refining programs to produce executable code; while Castor retains all forms of the program concurrently, something like the PDS method of separating versions into modules may be mandatory when large programs are involved.

The Incremental Program Construction component of the Carnegie-Mellon University Gandalf system is an interactive program development tool for Ada [Feiler 79]. The Cornell Program Synthesizer is a similar system for editing and interactively executing PL/CS programs (a dialect of PL/I) [Teitelbaum 81]. These systems each offer a source-language oriented editor for creating and modifying programs. The editor knows the syntax of the language and prompts the user with statement skeletons to be filled in. Since text is parsed and checked as it is entered, the resulting program

is necessarily syntactically correct. These systems also offer interactive debugging, permitting the user to make source language corrections and invoke incremental re-translation of the program.

Structure-oriented editors of this kind are of great interest in a Rapid Prototype Programming environment. Since editing operations are defined only in terms of the grammar of the language, the problem of dealing with syntactic errors is completely eliminated. This facilitates refinement transformations particularly, and such an editor capable of handling program attachments would be ideal. The lack of such an editor was found to be an inconvenience in Castor.

TRANSFORMATION SYSTEMS

Transformation systems are based on an approach summarized by Knuth:

> The programmer using such a system will write his beautifully structured, but possibly inefficient, program P; then he will interactively specify transformations that make it efficient. Such a system will be much more powerful and reliable than a completely automatic one. ... The original program P should be retained along with the transformation specifications, so that it can be properly understood and maintained as time passes. ...
>
> A "calculus" of program transformations is gradually emerging, a set of operations which can be applied to programs without rethinking the specific problem each time. [Knuth 74](page 283)

A broad catalog of such transformations has been compiled, addressing in particular the conditions of transformation validity [Standish 76a]. Work has also been done on chaining of transformations [Loveman 77, Kibler 78] and verification of transformation correctness [Kibler 78]. Consideration has also been given to the question of detecting areas of a program where transformations ought to be applied [Wegbreit 76].

DRACO is a transformation system which implements the Component Software approach described in Chapter 3 [Neighbors 81]. This system supports both refinement transformations and optimizing transformations. This involves the creation of large transformation data bases on an application domain basis. To reduce the amount of direction provided by the user, the system has default refinements that it can make, and the system will also suggest applicable transformations based on the most recent actions. Rules for chaining transformations are called "metarules" and are automatically generated by the system when a new transformation is added to the data base.

The Transformation Implementation (TI) project of Balzer <u>et al</u> is aimed at automating the refinement of executable programs from much more abstract program specifications [Balzer 79, Goldman 80]. Although such systems are intended ultimately to be fully automatic,

research is still being done to reduce the amount of human supervision required [Fickas 80].

In Castor, refinement is a purely manual process, although there is potential for coupling these technologies in the future to automate the refinement and optimization processes in a prototyping system. Transformational Implementation will be particularly desirable for prototyping to the extent that programming in high-level program specifications becomes practical.

VERY HIGH LEVEL LANGUAGES

Very high level languages are basically a continuation of the historical development of more and more powerful languages. SETL is a very high level language developed under the direction of J. Schwartz and is so named for its emphasis on the mathematical concept of sets [Dewar 78]. The basic data types are integers, reals, bit strings, and character strings. Structured data types are tuples, which are ordered collections of values, and sets, which are unordered collections of unique values. Maps are sets of ordered pairs which can be accessed using mathematical functional notation. Maps can also be multi-valued -- that is, they can be multi-valued relations. The language also provides control structures which iterate over sets or sequences using the predicate calculus quantifiers "exists"

and "for all."

The VERS2 language bears a number of similarities to SETL [Earley 74, Earley 75]. In VERS2 a distinction is made between tuples and sequences -- a tuple is an ordered collection of predetermined size consisting of heterogeneous elements, whereas a sequence is an ordered collection of unbounded size containing homogeneous elements. VERS2 also has sets and relations. The most interesting aspect of VERS2 is the formalization and unification of the "iterator" concept. An iterator is basically an implied loop over a data structure -- either over a set or a sequence. Conditions can be added to filter the iteration values, to control the termination of the iteration, and to combine iterations in various ways. So-called "iterative operators" are driven by iterators to do various things -- for example, replace elements, build sets or sequences, delete elements, check universal and existential quantification, or execute some general language statement. VERS2 is also endowed with a pattern matching capability in conjunction with iterators.

The primitives of languages such as these are suggestive of programming domains that could be made available in function libraries for Rapid Prototyping in specific problem modeling domains.

CHAPTER 8

CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK

RAPID PROTOTYPING

Rapid Software Prototyping is a stage in the software lifecycle where the attempt is made to squeeze all the remaining development stages into as short a time as possible. When applicable, this offers much faster feedback to the early phases of the lifecycle -- namely, the requirements analysis and specification phases. By doing this, wasteful pursuit of uncertain or incorrect goals can be avoided. The benefits from this are less wasted effort and expense in the "serious," production-quality development, and a final product that is more responsive to the users' true needs. The final program may also be better designed and implemented, by virtue of the added experience on the part of the implementors.

The key issue is therefore to determine just when a prototyping effort is called for and just what purpose the prototype is to serve. Prototyping is most attractive when the proposed system is radically new in some regard. The

system may be new because such a system has never been built or it may simply be that the specifiers and ultimate users have limited experience with that particular type of system or with automated systems in general. Rapid Prototyping is most valuable when there is uncertainty about the early stages of analysis that can only be resolved by experience with a working system.

The purpose of building the prototype should be clearly defined, since one of the most effective ways to speed prototyping is to leave out inessential functions. Since the prototype cannot reproduce faithfully all the properties of the final system, care must be taken in selecting what to implement. A prototype plan should identify both aspects of the system that are in need of confirmation as well as those in which there is adequate confidence and understanding. With these criteria clearly identified, the prototyping effort can be kept to the least possible scope.

Prototyping is a subject for which it is difficult to provide experimental validation. As in most software engineering studies, the most important phenomena are associated with large, expensive projects, and it is these projects that are the most difficult to produce in quantity for experimental purposes. It would be most unattractive (to anyone having to pay for it) to run two parallel developments of the same large project simply as an

experimental control on the use of prototyping. On the other hand, with the growing interest in Rapid Prototyping and the strong case for its use in selected applications, its seems likely that Rapid Prototyping may soon be officially incorporated into the procurement of some software systems.

CASTOR AS A PROGRAM DESIGN LANGUAGE

Castor is a language extension of Ada designed with Rapid Prototyping in mind. Calling forms give Castor the flexibility of a PDL, allowing program designers to sketch out their ideas and to move from problem to problem easily. Calling forms can later be refined by replacing them with suitable program fragments or by leaving the text as written and adding remote definitions. Because the design and implementation are both represented in the same data base, the transition from design to implementation can be continuous and can be managed entirely within the machine. In addition, the tools of design analysis are available for use during implementation -- this makes it convenient, for example, to get reports on calling forms not yet defined and on consistency of module interfaces. The problem of update consistency is reduced, but not eliminated, because broad changes need to be made consistently at all pertinent levels of refinement. Having all this information in one place

helps toward this end, however.

The use of macros in Castor allows calling forms to be used as data types, novel control structures, and variable names. This is in addition to their more conventional interpretation as procedure or function calls. Macro definitions themselves are written in Castor using built-in data types and functions for dealing with program fragments in their internal form. Program fragments are initially specified as character strings with indicated substitution points. A certain amount of overhead is incurred since the lexical analyzer and parser must be used when generating an internal node; in addition to this, a preliminary unparsing action is also required whenever an existing internal node is to be substituted into such a fragment. The benefit of this is that it makes macro definitions independent of the method of program representation, and the writer of macro definitions is spared from learning and remembering the many details of such a representation.

In Castor, program refinements are represented by attachments made at various points in the program. In general an attachment is associated with a phrase or a sequence of phrases of the program, and the value attached can in principle be anything. For Castor refinements the value attached consists of another program phrase or a sequence of phrases, an indication that the attachment is a

refinement, and the refinement name. This representation of refinements makes it possible to view the program at any previous state of refinement and to restore it to that state if necessary.

In the actual use of Castor a few problems arose. One of these was the need for a structure-oriented editor for managing program refinements. The LISP structure editor was pressed into service for Castor, and a few editor macros were added for convenience. The result was adequate, but hardly an ideal tool. Fortunately, interactive structure-oriented editors of this kind are under development and will certainly be important for Rapid Prototyping. In addition, it was found that explicit management of program attachments can be distracting and error prone. These should be implemented in an error-proof way as part of the editor.

Another area for future development is the construction of library packages and macro definitions for prototyping. Input/output in Ada is very basic, and the ability to direct input and output by some kind of grammar notation would be a great enhancement to the prototyping power of the language. Other packages for list handling and other high-level data structures such as relations, mappings, and sets would be most worthwhile. One particularly important concept would be to provide a systematic distinction between value and

pointer references -- for example, reflecting the difference between

```
X := The (Y) In (S) Such That (P(Y));
```

and

```
Let (X) Be In (S) Such That (P(X));
```

In the latter case only, the following would be a meaningful subsequent operation:

```
Delete (X);
```

An ample stock ot ideas and concepts is to be found in the very high level language literature, as we have discussed in Chapter 7.

THE ADA PROGRAMMING LANGUAGE

A number of difficulties with Castor as a prototyping language were caused by the characteristics of Ada. The choice of Ada was made attractive by Ada's potential to become a widespread standard. A language for writing programs that are portable to a large variety of systems is particularly attractive, especially when it also means that it is portable to the "understanding" of a large number of programmers. It is also desirable to have access to an ever-growing set of programming tools, as is expected to

develop for Ada environments.

Unfortunately for our purposes, when Ada was created a primary design goal was maintaining program integrity over long-term program life. It was stipulated from the beginning that the language designers might sacrifice the ease of writing programs to meet this end. In a way, however, this conflict is a good one from a research standpoint, since it points out the trade-offs most clearly.

An Ada "laundry" is a proposed mechanism for allowing programmers to relax the rules of the language and for automatically or at least interactively introducing the kinds of redundancy required by the pure language. As a future research area, such a tool could provide the following in support of Rapid Prototype Programming:

1. Introduction of intermediate type identifiers for compound data type declarations.

2. Declaration of the types of formal parameters of procedures and functions.

3. Construction of package specifications.

4. Ordering of the declarations in declarative parts to eliminate forward references, and introduction of subprogram specifications where necessary.

5. Transformation of a procedure definition to a function definition and vice versa.

6. Elimination of the verbiage associated with simple function definitions.

7. A facility for passing structured values between two points in a program without having to provide a type declaration (and possibly variable declarations) in a third place.

8. Implicit type declaration of identifiers by some convention -- for example: "Integer_1 := Y + Z;."

We note that many of these services cannot be completely automatic since the user should provide intelligible identifiers and other information.

SPECIFICATIONS

Another issue not addressed by the Castor system is the issue of formal specifications. Formal specification languages are currently under development, but they do not yet constitute an established technology [Goldman 80]. As we have noted, automatic refinement of languages of this sort is a very attractive prospect for Rapid Prototype Programming, if not for ordinary programming. However, much remains to be done before this will be achieved. Specification languages may still prove useful in prototyping before then by helping to control the transition from a prototype program to a later full-scale implementation. In this way the relationship between the prototype and the final system would be formally established.

BIBLIOGRAPHY

[Ada 80]
Reference Manual for the Ada Programming Language: Proposed Standard Document. United States Department of Defense, Jul 1980.

[Allen 76]
Allen, R.E., and D.A. Smith. "Functional Specification HMP-1632 Microprocessor." Hughes Aircraft Company, HMP-1670 ENB 5.1.1A, Aug 1976.

[Balzer 79]
Balzer, Robert. "Transformational Implementation: An Example." ISI Draft, Aug 1979.

[Boehm 73]
Boehm, Barry W. "Software and Its Impact: A Quantitative Assessment." Datamation, 19(52), May 1973, 48-59.

[Brooker 63]
Brooker, R.A., I.R. MacCallum, D. Morris, and J.S. Rohl. "The compiler-compiler." Annual Review of Automatic Programming, 3, 1963, 229-275.

[Caine 75]
Caine, Stephen H., and E. Kent Gordon. "PDL -- A tool for software design." in Software Design Techniques, P. Freeman and A.I. Wasserman (eds), IEEE Press, Catalog No. EHO 161-0, 1980.

[Cheatham 79]
Cheatham, Thomas E., Jr., Judy A. Townley, and Glenn H. Holloway. "A System for Program Refinement." Fourth International Conference on Software Engineering, Sep 1979, 53-62.

[Conrad 76]
Conrad, William E. "Rewrite User's Guide." Harvard University, Center for Research in Computing Technology, Apr 1976.

[Dewar 78]
Dewar, Robert B.K. "The SETL Programming Language." Computer Science Dept., Courant Institute, NYU, 1978.

[Dewar 79]
Dewar, Robert B.K., Arthur Grand, _et al_. "Programming by Refinement, as Exemplified by the SETL Representation Sublanguage." ACM Transactions on Programming Languages and Systems, 1(1), Jul 1979, 27-49.

[Earley 74]
Earley, Jay. "High Level Operations in Automatic Programming." SIGPLAN Symposium on Very High Level Languages, Mar 1974, 34-42.

[Earley 75]
Earley, Jay. "High Level Iterators and a Method for Automatically Designing Data Structure Representation." Journal of Computer Languages, _1_, 1976, 321-342.

[Feiler 79]
Feiler, Peter H. "IPC System Version 1: Incremental Program Construction." Carnegie-Mellon University, Department of Computer Science, 1979.

[Fickas 80]
Fickas, Stephen. "Automatic Goal-Directed Program Transformations." First National AAAI Conference, Stanford, 1980.

[Freeman 80]
Freeman, Peter, and Anthony I. Wasserman (eds). _Tutorial on Software Design Techniques_. IEEE Press, IEEE Catalog No. EHO 161-0, 1980.

[Goldberg 75]
Goldberg, P.C. "Automatic Programming." _Programming Methodology_, Goos and Hartmanis (eds), Lecture Notes on Computer Science, _23_, Springer Verlag, 1975, p 347.

[Goldman 80]
Goldman, Neil M., and David S. Wile. Gist Language Description. Information Sciences Institute, Nov 1980.

[Hart 81]
Hart, Hal. "Ada For Design: An Approach For Transitioning Industry Software Developers." NSIA Software Group Conference, Alexandria, VA, Oct 1981.

[IBM 72]
IBM Corp. "OS Assembler Language." Order No. GC28-6514-8.

[Kibler 78]
Kibler, Dennis F. Power, Efficiency, and Correctness of Transformation Systems. Ph.D. Thesis, ICS Department, UC Irvine, 1978.

[Knuth 71]
Knuth, Donald E. "An Empirical Study of Fortran Programs." Software -- Practice and Experience, 1(2), Jun 1971, 105-133.

[Knuth 74]
Knuth, Donald E. "Structured Programming with go to Statements." Computing Surveys, 6(4), Dec 1974, 261-301.

[Krieg 80]
Krieg-Brueckner, Bernd, and David C. Luckham. "ANNA: Towards a Language for Annotating Ada Programs." Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language, Boston, 128-138, Dec 1980.

[Leavenworth 66]
Leavenworth, B.M. "Syntax Macros and Extended Translation." CACM, 9(11), Nov 1966, 790-793.

[Loveman 77]
Loveman, David B. "Program Improvement by Source-to-Source Transformation." JACM, 24(1), Jan 1977, 121-145.

[Martin 74]
Martin, W.A., M.J. Ginzberg, R. Krumland, B. Mark, M. Morgenstern, B. Niamir, and A. Sunguroff. Internal memos. Automatic Programming Group, MIT, 1974.

[Meehan 79]
Meehan, James R. (ed). The New UCI LISP Manual. Lawrence Erlbaum Associates, 1979.

[Meehan 80]
Meehan, James R. The UCI MLISP Reference Manual. ICS Department, UC Irvine, Mar 1980.

[Meehan 81]
Meehan, James R. F Editor Manual. ICS Department, UC Irvine, Jan 1981.

[Neighbors 81]
Neighbors, James Milne. Software Construction Using Components. PhD Thesis, ICS Department, UC Irvine, Tech. Report 160, 1981.

[Naur 63]
Naur, Peter (ed). Revised Report On The Algorithmic Language Algol 60. CACM, 6(1), Jan 1963, 1-17.

[Ramamoorthy 79]
Ramamoorthy, C.V., and Raymond T. Yeh. Tutorial: Software Methodology. Chicago: Palmer House, IEEE Catalog No. EHO 142-0, 1979.

[Smith 79]
Smith, David A., and Thomas A. Standish. "Research on Interactive Program Manipulation: Final Report." ICS Department, UC Irvine, Tech. Report, Dec 1979.

[Standish 76]
Standish, Thomas A., Dennis F. Kibler, and James M. Neighbors. "Improving and Refining Programs by Program Manipulation." ACM National Conference, 1976, 509-516.

[Standish 76a]
Standish, T.A., D.C. Harriman, D.F. Kibler, and J.M. Neighbors. The Irvine Program Transformation Catalogue. ICS Department, UC Irvine, Jan 1976.

[Standish 80]
Standish, Thomas A. "ARCTURUS: An Advanced Highly-Integrated Programming Environment." Software Engineering Environments, Proceedings of the Symposium held in Lahnstein, Federal Republic of Germany, Jun 1980, 49-60.

[Stoneman 80]
Requirements for Ada Programming Support Environments: "Stoneman." United States Department of Defense, Feb 1980.

[Taylor 81]
Taylor, Tamara, and Thomas A. Standish. "Initial Thoughts on Rapid Prototyping Techniques." To appear in proceedings of the ACM-SIGSOFT Second Software Engineering Symposium: Workshop on Rapid Prototyping, Apr 1982.

[Teitelbaum 81]
Teitelbaum, Tim, and Thomas Reps. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment." CACM, 24(9), Sep 1981, 563-573.

[UCI Workshop 78]
Proceedings of the Irvine Workshop on Alternatives for Environment, Certification, and Control of the DoD Common High Order Language. ICS Department, UC Irvine, Jun 1978.

[VanHorn 80]
Van Horn, E. "Software Must Evolve." Workshop on Software Engineering, Academic Press, 1980.

[Waugh 80]
Waugh, D.W. "Ada As A Design Language." IBM Software Engineering Exchange, 3(1), Oct 1980, 8-12.

[Wegbreit 76]
Wegbreit, Ben. "Goal-Directed Program Transformation." IEEE Transactions on Software Engineering, SE-2(2), Jun 76, 69-80.

[Wells 72]
Wells, Mark B., and James B. Morris. "The Unified Data Structure Capability in Madcap VI." International Journal of Computer and Information Sciences, 1(3), 1972, 193-208.

[Wirth 71]
Wirth, Niklaus. "Program Development by Stepwise Refinement." CACM, 14(4), Apr 1971, 221-227.

[Workshop 82a]
ACM-SIGSOFT, Software Engineering Symposium: Rapid Prototyping, preprint papers, Apr 1982.

[Workshop 82b]
Proceedings of the ACM-SIGSOFT Second Software Engineering Symposium: Workshop on Rapid Prototyping, to be published.

[Zelkowitz 79]
Zelkowitz, Marvin V., Alan C. Shaw, and John D. Gannon. Principles of Software Engineering and Design. Prentice-Hall, 1979.

APPENDIX A

AN EXAMPLE OF RAPID PROTOTYPING

In this appendix we show how Rapid Prototyping can be used to develop a realistic software system of modest size. The proposed system is for automating many of the data structures and operations found in a business office. In particular, we suppose that as system designers we wish to explore the use of a desk-top computer to perform functions normally done on paper.

Office automation is an active area of development and competition in the business community today, and it is not the aim of this thesis to make a contribution in this area. Rather, we use this domain to show the prototyping process in action. This is an ideal case for our demonstration since it is a fresh and evolving application area. This example also has the expository advantage of being more or less universally familiar, so we need not define and explain the motivation for the basic concepts of the application domain.

In the following we described the steps that were used in designing and implementing an electronic office prototype

program using the Castor system. The steps used in the prototyping process are as follows:

1. The objectives of the prototype are identified. This is to answer the question "Why is a prototype of this system desirable?" This also establishes criteria for measuring the success of the prototyping effort.

2. Some broad choices are listed for the functions to be offered by the target system.

3. A Model System is described; the document doing this is basically a prototype <u>system specification</u> for the target system. Not all of these functions need to be in the prototype -- rather, this is another choice space which is more limited and is described in greater detail.

4. Functions to be implemented in the prototype are selected. For the electronic office it appeared that some concepts could be evaluated independently of the actual set of office functions performed. Thus, two prototypes were proposed, only the first of which is considered in this appendix.

5. The top level of the prototype program is characterized.

6. A modular design of the prototype is given.

7. The design is refined in a stepwise manner, using the bookkeeping of the Aide to monitor progress toward a complete implementation.

OBJECTIVES OF THE PROTOTYPE

To begin with, we consider the features we wish to prototype and what we wish to determine from the prototype. The object of our investigation is the use of a desk-top computer with the characteristic of offering exclusive and

continuously on-line access to a local data base. We may suppose we have access to other systems (including other desk-top systems of this sort) via some kind of data network. We also assume we have a high-speed video display and either local or remote printing capability.

Beyond these basic assumptions there are a great many design choices available. Rather than trying to make the "best" decision for each choice, the prototyping approach is simply to proceed quickly, making what seem to be reasonable trade-offs and design choices. No amount of careful planning and design, for example, will tell us with certainty if a certain two-dimensional input device such as a mouse or joy-stick will be perceived by users as a valuable capability. This depends on how use of the system evolves, and this can only be determined by letting users gain experience with it. In fact we don't even know if the computer itself is going to be valuable for the purposes we are exploring -- this is why we are building the prototype.

Clearly anything we might wish to do with a desk-top computer could alternatively be done on a timesharing system with a desk-top terminal. The most important differences offered by the desk-top computer would be in having continuous on-line access, having exclusive access to the processor, and possibly in having higher data rates for information display. These properties make the desk-top

computer attractive for uses which would seem frivolous if it took time, effort, and a recurring expense to log onto a remote system in order to use them.

Rather than build a hardware prototype of this system, however, we will emulate its behavior by using prototype software on a timesharing system. This will be representative of the actual system, if

1. We can remain logged-in continuously during periods of evaluation.
2. The display speed is comparable to the display speed of the dedicated system.
3. The performance of the software and the response offered by the timesharing system are good enough to be representative of a dedicated system.

We can use the features of the prototype host system to simulate other devices which would be reachable through the proposed data network, such as other users' desk-top systems or a community printing device.

A SET OF FUNCTIONS TO SELECT FROM

In this section we consider a broad set of functions that the proposed system might offer. Here, in the briefest possible fashion we explore various possibilities for the proposed system, and from this list a few interesting and representative capabilities will be chosen for the prototype. The following items are data structures found in

an office which are generally found on paper rather than in an electronic medium:

1. Telephone or address directories
2. Work agendas (ie, what to do today, what to do for this project, etc.)
3. Meeting agendas
4. An appointment calendar
5. Reference lists (bibliographies), with provision for noting books or articles loaned out
6. Project schedules (showing milestone goals and progress)
7. Organization charts
8. Memos of various sorts (notebook entries, proposals, reports)
9. Forms of various sorts (meeting notices, request forms)
10. Help files, account numbers, and passwords for various computer systems
11. Time cards
12. Backups of files on various computer systems
13. Vacation schedules and history
14. A dictionary or spelling guide
15. Mail
16. Mail distribution lists

We note that of these, the last three have some counterpart in contemporary computer systems. Spelling corrector programs serve one of the purposes of a dictionary, though they are not used in exactly the same way. Electronic mail

programs have also become quite popular and sophisticated.

The user may wish to create such data structures, edit them, send them to other people as mail, and print them. There are also typical file-system operations such as listing directories of files, renaming or regrouping files, deleting files, and so on.

Special purpose operations may also be desired such as searching the calendar for free appointment times, looking up an individual's phone number or address, searching a bibliography for key words, or answering a piece of mail. In addition to these, some real-time functions may be desirable -- for example, requesting an alarm clock setting for a selected appointment in the appointment calendar, having the system notify the user when new mail arrives, or having messages sent or file protections changed on a given time or date.

Security must also be provided. A password or other locking mechanism is needed so that the user can leave the computer unattended with confidence. Confidential files must also be protected from unauthorized access over the data network.

THE MODEL SYSTEM

The next step is to develop the concepts upon which the proposed system is to be based. This may be a formal specification of the proposed system, it may be a brief high-level description of the prototype to be built, or it may be somewhere in between these extremes of scope and detail. In the present case we describe a superset of the functions we will prototype. This "Model System" allows us to explore various system concepts without committing ourselves to implementing them. In a sense this description serves both as a prototype of the System Specification and also as a menu of choices for the prototype we are to implement. In the present case this document is somewhat lengthy, but it is included here in the following pages to indicate the level of detail which seems appropriate at this point:

Basic Concepts

In the Model System the basic unit of data managed is the document. Every document has a header and some contents. Each header is a list of attribute-value pairs which may include a document-form type, a creation date, a level of protection, and a document name. Each of these is considered optional, including the document name.

Other possible attributes might be keyword indices, subject, author, and various kinds of document status. Although a full implementation might offer various views of a header, both detailed and brief, our prototype will always display the whole header and will provide support only for a limited set of attributes.

The form or template of a document allows the editor to supply various information fields literally or with default or automatically generated values. Some information can be protected from changes, like the parts which would be pre-printed on a paper form. In a space-efficient implementation these fields might not even be stored and would be furnished only on print-outs by programs knowing the form type. Fields may be flagged for validity checks -- for example, checking that dates are valid, or that phone numbers are correct or at least have the correct form, or that names of individuals are spelled correctly.

Documents may be grouped into document files, within which they have a sequential order. A file then is like a filing folder in which related documents are collected. We do not require in general that all documents in a file be of the same kind. A file may also contain other files -- this is analogous to a file drawer containing file folders, a file cabinet containing file drawers, and so on. The nesting of files may continue to a reasonable depth. A given document file may be composed of both documents and nested files of varying depths, and it may have zero length. A node is an item in a file, whether it be a document or another file. The overall system structure therefore consists of a single file at the top level. Files also have attribute lists, generally containing at least a file name.

In addition to this upward hierarchy, we also permit some documents to be sequences of similar items -- for example, days in a calendar or different entries in a bibliography. If these repeated units are instances of a document template, then this is called an iterated template document.

In addition to documents and files, the system will also contain programs and template descriptions. For the prototype we will define a fixed set of forms and programs. In a full-scale system, extensibility of this set of programs and forms would, of course, be an area of particular interest.

File Management Commands

The syntax for commands dealing with files is as follows:

```
QUAL          = NAME | NUMBER | -NUMBER
NODE          = QUAL {. QUAL}
FILE          = file_NODE
DOC           = document_NODE
NODE_SET1     = NODE                                   |
                FILE . *                               |
                FILE . (QUAL : QUAL)
NODE_SET      = NODE_SET1 {+ NODE_SET1}
TOKEN         = NAME                                   |
                NUMBER                                 |
                - NUMBER                               |
                .                                      |
                :                                      |
                ,
COMMAND       = "PRINT" NODE_SET ;                     |
                "DIRECTORY" NODE_SET ;                 |
                "APPEND" FILE "=" NODE_SET ;           |
                "APPEND_COPY" FILE
                   "=" NODE_SET ;                      |
                "DELETE" NODE_SET ;                    |
                "CREATE" DOC [form_NAME] ;             |
                "EDIT" [DOC] ;                         |
                "CLOSE" DOC ;                          |
                "LOOK" DOC {Arbitrary_TOKEN} ;         |
                "FIND" DOC {Arbitrary_TOKEN} ;         |
                "UNLOCK" Arbitrary_NAME ;              |
                "LOCK" ;                               |
                "SET_LOCK" Arbitrary_NAME ;
```

In this grammar notation, CAPITAL letters denote nonterminals, {braces} denote repetition zero or more times, [brackets] denote optional items, and vertical bar (|) denotes alternative constructs. Quotes are placed around a terminal which would otherwise look like a nonterminal or a meta-symbol ("=" or "MOVE"). The use of a lower case word as a prefix to a nonterminal indicates that the entity specified must semantically turn out to be an instance of the item mentioned in lower case letters. For example a FILE is defined to be a file_NODE -- therefore it must be a

NODE which turns out to be a file (and hence not a document).

The semantics of these commands can be described briefly. A qualifier selects a node from the nodes of a file. A name may be used only if the node has a name attribute; otherwise a number may be used denoting the node's position in the file, or a negative number may be used denoting the node's position from the end of the file. A NODE_SET1 is an expression whose value is a sequence of nodes, either given by naming a single node, by naming all the nodes in a file, "FILE.*," or by naming a subrange of the nodes in a file, "FILE.(QUAL:QUAL)." A NODE_SET is a similar value which may be augmented by concatenating many such expressions.

The file commands are largely self-explanatory. PRINT generates a hard copy of all the documents in the specified node set. DIRECTORY displays a list of all document headers within the specified set of nodes; note that the documents in a file expression are listed by their fully qualified names for this purpose. APPEND creates or appends to a file. Documents or files appended to a file are deleted from their original location. APPEND_COPY is like APPEND, but does not delete the original copy of the documents or file. CREATE is for creating a new document: the system enters a mode where document editing commands are recognized, and if a form name is specified then the appropriate document template is invoked. EDIT specifies an existing document and enters edit mode accordingly.

CLOSE invokes the form-specific checking appropriate to a given document -- this is only necessary if editing of the document was suspended without closing it. LOOK is another form-specific operation -- for example, for looking up specific information in a calendar, dictionary, or directory. FIND, like LOOK, is a form-specific search in a document. In general, if LOOK has an ambiguous reference it will display the first matching occurrence. FIND, on the other hand, will display such occurrences one at a time. An empty command line will cause FIND to sequence to the next instance. After a LOOK or in the midst of a FIND, an EDIT command with no document specified will cause the editor to be invoked on the current document at the current location.

The LOCK command places the system in "locked mode" where the only command it will accept is an appropriate UNLOCK command. A password must be provided to unlock the system. The SET_LOCK command is for changing this password.

Document Templates

In order to impose structure on some documents and allow the system to furnish defaults and perform other special operations, it is necessary to define the template of a document. The template describes the format of the document or of repeated items in a document. A template contains five kinds of information: inviolate text, default text, default function names, comments (prompts), and initial text. The notation given below indicates how these different kinds of text appear when the template is first invoked:

Inviolate text	eg, <u>Author</u>
Default text	eg, \Prototyping Project\
Default function name	eg, \(Todays_Date)
Comment or prompt	eg, \<Author's name>
Initial text	eg, This just looks like text.

Fields in the template are delimited by inviolate text. In the prototype, fields must be rectangular in shape. All the fields present on a given line may be vertically lengthened by inserting a blank line containing appropriate field separators. Any insertion made over comments or defaults causes the comments or defaults to disappear. There is no distinction made between initial text and text which is inserted during editing: initial text may be overwritten or explicitly cleared. Closing a document automatically causes any remaining defaults to be instantiated and comments to be removed.

Editor Commands

The editor in the Model System is patterned after the F editor written by Jim Meehan of the University of California at Irvine [Meehan 81]. This is a full screen editor to which features are added for dealing with document fields and defaults. By using the framework of an established editor we are less concerned with making sure that all conventional editor functions have been provided for. This also enables us to take advantage of the existing overall design of such an editor and of the algorithms it uses. The editor in the prototype is a modification of an F-like editor written in Ada by Scott Ogata.

Each editor command is entered as a control character. For example, Control-Z closes the document and leaves edit mode, while Control-X leaves edit mode without doing the form-specific CLOSE operation. It is also possible to close the document but remain in the editing session -- this permits the user to preview the final document. The

following are conventional full-screen editing operations:

Key	Function
^S	Delete One Character
^F	Delete One Line
^T	Move Window +10 Lines
^W	Move Window - 10 Lines
^P	Skip To End Of Document
^V	Skip To End Of Line
^X	Leave Edit Without Closing
^Z	Close And Leave Edit Mode
^\	Move Cursor Right
^H (BS)	Move Cursor Left
^_	Move Cursor Up
^J (LF)	Move Cursor Down

A number of edit commands specific to our proposed electronic office system are entered as two control characters: a Control-B which serves as an escape, followed by an appropriate control character. These operations are specific to the electronic office system:

Key	Function
^B^N	Skip To Next Field
^B^P	Skip To Previous Field
^B^X	Clear Field And Skip To Next
^B^I	Invoke Another Template Instance
^B^F	Instantiate Field Default
^B^Z	Perform Close But Continue Editing

Alarm Functions

In the Model System the only timed functions are alarms associated with the appointment calendar. An entry in the calendar may be flagged with an asterisk, indicating that an alarm at that time is desired. When the calendar is closed, the timer queue is cleared and then all alarm requests found in the calendar are entered (or re-entered) in the new timer queue. Thus to cancel an alarm the user simply edits the calendar and deletes the selected alarm request.

When an alarm occurs, an audible tone sounds at the terminal and a message is displayed, consisting of the calendar entry causing the alarm and the immediately following calendar entry, if there is one. In this way, the user can request an alarm as a reminder some interval of time before a particular appointment.

Document Forms In The Model System

The Model System includes only a few of the proposed document forms. These are chosen with the following criteria in mind:

1. To demonstrate a variety of applications of the system.

2. To show the usefulness of basic concepts such as template filling and files of documents.

3. To try to find uses which seem most attractive for prospective users, such as managing agendas or telephone numbers.

The following formats are described as part of the Model System:

1. Agenda
2. Appointment Calendar
3. Telephone Directory
4. Bibliography

The formats and special operations for these documents are given in the remainder of this section.

Agendas An agenda is a document which lists brief items in an order of priority. Priority is indicated by numeric labels, and completed items may be retained with a nonnumeric label such as "xx." Numeric labels are given in decimal with an integer part, a decimal point, and an optional fractional part.

Functions peculiar to agendas are as follows:

Close -- At the end of an edit (or any time the document is "Closed"), the agenda items are sorted in order of increasing priority. Completed items (with non-numeric labels) are moved to the bottom, preserving their relative order. Numbered items are then renumbered starting at 1 in increments of 1. Items with equal priority retain equal priority and their same relative order.

Appointment Calendars The following illustrates the template for an appointment calendar (truncated on the right to fit on this page):

```
 SUN\<date> MON\<date> TUE\<date> WED\<date>  ...
---------------------------------------------  ...
⊥\<appts>  ⊥\<appts>  ⊥\<appts>  ⊥\<appts>  ⊥ ...
```

The primary data is stored in seven vertical fields, each of which is eleven characters wide. Each column, naturally, has appointments for one day of a given week. Multiple instances of this template are for successive weeks. The long field of inviolate hyphens delimits the tops of the day fields, and the inviolate vertical bars separate adjacent day fields. The last line of the template can be replicated to lengthen the fields vertically.

Each appointment begins with a digit (or an asterisk followed by a digit) in the leftmost column of a vertical field. This digit begins a time specification. This time specification may stand alone or may be followed by a hyphen and another time specification. The asterisk, if specified, indicates that an alarm is to be generated at the time specified (or at the first time, if two are given). The rest of the information about the appointment is arbitrary and continues until the next appointment or the end of the vertical field. The following is an example:

```
 SUN Jun13  MON Jun14  TUE Jun15  WED Jun16    ...
---------------------------------------------  ...
⊥          ⊥8-9 Status⊥          ⊥*1030 Call⊥ ...
⊥          ⊥  Review  ⊥          ⊥  IRS     ⊥ ...
⊥          ⊥          ⊥*12 Lunch ⊥  Auditor ⊥ ...
⊥          ⊥          ⊥   Appt   ⊥          ⊥ ...
```

The meaning of a time specification depends on the number of digits. One or two digits specify an hour; three or four digits specify an hour and a minute. The time may be followed by a letter "A" for AM or "P" for PM -- if these are omitted the default time is between 600A (6:00 AM) and 559P (5:59 PM).

Functions particular to an appointment calendar are as follows:

Close -- This sorts the appointments for each day in increasing order and puts a blank line between appointments. This also causes the timer queue to be emptied and then causes the document to be scanned, enqueueing (or re-enqueueing) each alarm request that is found.

Look -- The user specifies a date. This causes the week containing that date to be displayed. The date is specified with a three-letter month abbreviation and a decimal day number, either as "day month" or "month day."

Find -- The user specifies a lower bound and an upper bound time in standard form, separated by a hyphen. This request causes the calendar to be searched for a free appointment time within the specified time-of-day range. The search starts with the current day.

Telephone Directories A telephone directory is a list of names with telephone numbers and possibly other information. A typical line has the form

Lastname, Firstname (Alias) number number ... number

The numbers may have any form, and there may be comments interspersed. Only the name fields of the line are interpreted.

The operations associated with directories are:

Close -- Sorts the document on the names.

Look -- Looks up the entry associated with the specified name and displays the line found on the terminal. In case of ambiguous reference, all selected lines are displayed. The name specified may have one of the following forms: 1) "Lastname," 2) "Firstname," 3) "Alias," 4) "Firstname Lastname," 5) "Alias Lastname."

Find -- Looks up entries which match a given name.

Bibliographies A bibliography item consists of a bibliography name abbreviation, a series of index words or keywords, an author, a title, publication information, and a comments field. The comments extend from the end of the publication information until the next item or the end of the document. The entry abbreviation is identified by the fact that it is enclosed in brackets. The following is an example:

[Brooks 79] \Software Engineering\Management\
Brooks, Frederick P.
The Mythical Man-Month.
Addison-Wesley, 1979.
Loaned to John Doe -- Dec 5, 1981.

The operations particular to a bibliography are as follows:

Look -- Looks up an item or items by abbreviated name or by index keyword.

Find -- Searches for an item or items matching a particular keyword.

FUNCTIONS SELECTED FOR THE PROTOTYPE

Having described the Model System, the next step is to determine how much of this system is to be implemented in the prototype. With this in mind we establish (somewhat arbitrarily) the following goals for the prototype:

1. It should help us evaluate the "file" and "document" concepts and the command language facilities for dealing with them.

2. It should help us evaluate the "template" concept as supported by the editor. This includes handling fields, inviolate text, and field comments (prompts).

3. It should offer one or more of the document forms described in the Model System. For simplicity in the first version of the prototype, the only document form implemented was the Appointment Calendar.

4. Given the achievement of the above goals, a second iteration of the prototyping process would provide a large number of the office document forms and operations described above and would be used to evaluate the true usefulness of a desk-top office system.

Given the above goals, the design and implementation of the prototype now proceeds in a top-down manner. Because Castor is both a Program Design Language and a programming language, the distinction between these two phases is not well marked.

It would require too much space to present the entire prototype here and would require even more to show it in its entirety at each stage of development. In the remaining

sections of this appendix, therefore, we present only selected steps of this development in order to illustrate the refinement process and demonstrate various features of Castor and the Aide.

THE TOP LEVEL OF THE PROTOTYPE

The first representation of the prototype is a top-level procedure called "Office." The following excerpt of a Castor session shows this procedure.

```
$pp("office");
-- Refinements:

PROCEDURE Office IS
  Locked: Boolean := False;
BEGIN
  LOOP
     BEGIN
        Get User Command;
        Interpret User Command;
     EXCEPTION
        WHEN OTHERS =>
           Issue Error Message;
     END;
  END LOOP;
END Office;

$def_check;
office
  PROCEDURE get_user_command
  PROCEDURE interpret_user_command
  PROCEDURE issue_error_message
```

In this protocol, "$" is the system prompt character, and the command

```
pp("office");
```

is a command which causes the procedure "Office" to be pretty-printed. A list of current refinements in effect is printed (in this case the list is empty), and the procedure is then printed. The command

```
def_check;
```

checks the last-printed procedure ("Office") for undefined types, procedures, variables, and so on. The resulting list shows those procedure calling forms which are undefined at this time.

The next step is to introduce the refinement OFFICE1, the effect of which is shown by the following:

```
$show("office1");
$pp("office");
-- Refinements:  --OFFICE1

PROCEDURE Office IS
  --2(OFFICE1)
  TYPE Commandty IS (Print, Directory, Append, Del, Create,
     Edit, Close, Look, Find, Unlock, Lock, Set_lock);
  Command: Commandty;
  --
  Locked: Boolean := False;
BEGIN
  LOOP
     BEGIN
        --2(OFFICE1) Get User Command;
        ---          Interpret User Command;
        Get (Command);
        CASE Command IS
           WHEN Print =>
              Printcom;
           WHEN Directory =>
              Directorycom;
           WHEN Append =>
```

```
            Appendcom;
         WHEN Del =>
            Deletecom;
         WHEN Create =>
            Createcom;
         WHEN Edit =>
            Editcom;
         WHEN Close =>
            Closecom;
         WHEN Look =>
            Lookcom;
         WHEN Find =>
            Findcom;
         WHEN Unlock =>
            Unlockcom;
         WHEN Lock =>
            Lockcom;
         WHEN Set_lock =>
            Set_lockcom;
         WHEN OTHERS =>
            RAISE Error;
      END CASE;
      --
   EXCEPTION
      WHEN OTHERS =>
         Issue Error Message;
   END;
  END LOOP;
END Oftice;

$def_check;
office
  EXCEPTION error
  PROCEDURE appendcom
  PROCEDURE closecom
  PROCEDURE createcom
  PROCEDURE deletecom
  PROCEDURE directorycom
  PROCEDURE editcom
  PROCEDURE findcom
  PROCEDURE issue_error_message
  PROCEDURE lockcom
  PROCEDURE lookcom
  PROCEDURE printcom
  PROCEDURE set_lockcom
  PROCEDURE unlockcom
```

The "show" command directs that the state of the program

both before and after the OFFICE1 refinement are to be displayed when the program is pretty-printed. As shown, the refinement OFFICE1 has introduced two declarations (the type "Commandty" and the variable "Command") and has replaced the two statements in the loop body with two new statements: a call to procedure "Get" and a large CASE statement. At this point we have a larger set of undefined procedure calling forms and the exception "Error" is also found to be undefined.

A REFINEMENT OF THE PROTOTYPE

The next step of interest in this development is the introduction of a package defining most of the unresolved procedures in the main program. This is done by introducing the package as a separate compilation unit and placing a reference to this in the procedure "Office." This kind of reference is called a "body stub" and is Ada's facility for supporting top-down programming. Below we show the procedure "Office" once again with two refinements in effect: OFFICE1 and COMMANDS. The notation "--" associated with the COMMANDS refinement indicates that the program text is shown before and after that refinement, while all we see is the final form of the OFFICE1 refinement.

```
$show("commands");
$pp("office");
-- Refinements: OFFICE1 --COMMANDS

PROCEDURE Office IS
  TYPE Commandty IS (Print, Directory, Append, Del, Create,
     Edit, Close, Look, Find, Unlock, Lock, Set_lock);
  Command: Commandty;
  Locked: Boolean := False;
  --1(COMMANDS)
  PACKAGE BODY Offcom IS SEPARATE;
  --
BEGIN
  LOOP
     BEGIN
        Get (Command);
        CASE Command IS
           WHEN Print =>
              Printcom;
           WHEN Directory =>
              Directorycom;
           WHEN Append =>
              Appendcom;
           WHEN Del =>
              Deletecom;
           WHEN Create =>
              Createcom;
           WHEN Edit =>
              Editcom;
           WHEN Close =>
              Closecom;
           WHEN Look =>
              Lookcom;
           WHEN Find =>
              Findcom;
           WHEN Unlock =>
              Unlockcom;
           WHEN Lock =>
              Lockcom;
           WHEN Set_lock =>
              Set_lockcom;
           WHEN OTHERS =>
              RAISE Error;
        END CASE;
     EXCEPTION
        WHEN OTHERS =>
           Issue Error Message;
     END;
  END LOOP;
END Office;
```

The first version of the package "Offcom" is given below. This shows the flexibility of Castor calling forms as a formal program design language as the individual command operations are broken down into constituent actions.

```
$pp("offcom");
-- Refinements: OFFICE1 --COMMANDS
--1(COMMANDS)

SEPARATE (Office)
PACKAGE BODY Offcom IS

  PROCEDURE Printcom IS
  BEGIN
     For (N) In (Get Nodeset) Loop
        BEGIN
           For Each Document (D) In (N) Loop
              BEGIN
                 Print Document (D);
              END;
        END;
  END Printcom;


  PROCEDURE Directorycom IS
  BEGIN
     For (N) In (Get Nodeset) Loop
        BEGIN
           For Each Document (D) In (N) Loop
              BEGIN
                 Type Document Name Of (D);
              END;
        END;
  END Directorycom;


  PROCEDURE Appendcom IS
     F: Datanode;
  BEGIN
     F := Get File Node;
     IF Next Token /= Eq_token THEN
        RAISE Error;
     END IF;
     For (N) In (Get Nodeset) Loop
        BEGIN
           Appendl (N) To (F.Fil_val);
           Delete Node (N);
```

```
      END;
END Appendcom;


PROCEDURE Deletecom IS
BEGIN
   For (N) In (Get Nodeset) Loop
      BEGIN
         For Each Document (D) In (N) Loop
            BEGIN
               Delete External (D.Doc_externalname);
            END;
         Delete Node (N);
      END;
END Deletecom;


PROCEDURE Createcom IS
   D: Datanode;
BEGIN
   D := Get New Node;
   D.Kind := Doc;
   D.Externalname := New External Name;
   Load Form Buffer (D.Doc_form);
   Initialize Core Buffer;
   Editor;
END Createcom;


PROCEDURE Editcom IS
BEGIN
   D := Opt Get Node;
   IF Is Empty (D) THEN
      Editor;
   ELSE
      Load Document (D);
      Editor;
   END IF;
END Editcom;


PROCEDURE Closecom IS
BEGIN
   Load Document (Get Node);
   Close Document (D.Doc_form);
END Closecom;


PROCEDURE Lookcom IS
BEGIN
   Load Document (Get Node);
```

```
      Look (Get Search Keys);
   END Lookcom;


   PROCEDURE Findcom IS
   BEGIN
      Load Document (Get Node);
      Find (Get Search Keys);
   END Findcom;

BEGIN
   NULL;
END;
--


$def_check;
office
   EXCEPTION error
   PROCEDURE issue_error_message
   PROCEDURE lockcom
   PROCEDURE set_lockcom
   PROCEDURE unlockcom
offcom
   EXCEPTION error
   TYPE datanode
   TYPE OF d
   TYPE OF doc
   TYPE OF eq_token
   TYPE OF n
   PROCEDURE append1_to
   PROCEDURE close_document
   PROCEDURE delete_external
   PROCEDURE delete_node
   PROCEDURE editor
   PROCEDURE find
   PROCEDURE for_each_document_in_loop
   PROCEDURE for_in_loop
   FUNCTION get_file_node
   FUNCTION get_new_node
   FUNCTION get_node
   FUNCTION get_nodeset
   FUNCTION get_search_keys
   PROCEDURE initialize_core_buffer
   FUNCTION is_empty
   PROCEDURE load_document
   PROCEDURE load_form_buffer
   PROCEDURE look
   FUNCTION new_external_name
   FUNCTION next_token
   FUNCTION opt_get_node
```

```
PROCEDURE print_document
PROCEDURE type_document_name_of
```

In this example calling forms are used freely. Some appear in several places, while others are used only once. They are used as procedure calls -- for example:

```
Type Document Name Of (D);
```

as function calls:

```
F := Get File Node;
```

and as novel control structures:

```
For Each Document (D) In (N) Loop
   BEGIN
      Print Document (D);
   END;
```

Note that the latter control structure is intended to be a traversal of the leaves of a subtree in the file data structure. This calling form can only be formally implemented by a macro capability like that of Castor.

The definition check list shows that only five undefined identifiers remain in the main procedure "Office," but that a large number have been introduced in the package "Offcom." A number of implicit assumptions have been made concerning the representation of files and documents, and this is reflected in the large number of identifiers for which types are not known. The program text assumes that

"Datanode" is a record type, but the fields used in the text do not appear because "Def_check" does not handle field identifiers.

A number of calling forms are used in the prototype which might be part of a general list-processing package. "Appendl (-) To (-)," "For (-) In (-) Loop (-)," and "Is Empty (-)" are three such calling forms which are used in this package. No such library is in fact provided, and so these definitions must be furnished as part of the prototype.

MODULARIZATION OF THE PROTOTYPE

While not a great deal of detailed programming effort has been expended up to this point, enough is now known of the structure of the problem to give a modular breakdown of the functions that need to be provided. These are as follows:

1. The general purpose list functions (Offlst)
2. The parser (Offpar)
3. The editor (Offedi)
4. Utility functions or subroutines of Offcom (Offutl)

In building the actual prototype it was found later that the lexical rules are sufficiently complex to warrant a separate module:

5. Lexical Analyzer (Offlex)

When this program was written, the remainder of the development proceeded in the same manner shown above, using the Aide to provide an agenda of undefined calling forms at each step. Given the general modularization above, the major design task consisted of assigning functions to the appropriate modules and deciding on the conventions to be used in interfacing with the parser and the editor. The actual order of the refinements performed is indicated by the following list of refinement names:

OFFICE1:	The initial top-level program.
COMMANDS:	The package "Offcom."
OFFICE2:	The exception "Error" and printing of the error message.
PARSE:	The package "Offpar."
LEX:	The package "Offlex."
COMMANDS2:	Update preliminary assumptions concerning interfacing with the parser and lexical analyzer, and add the Lock and Unlock mechanism.
PARSE2:	The basic data structure for representing file and document structures.
EDIT:	The package "Offedi."
COMMANDS3:	The data structure of PARSE2 is modified to handle deletion. (This problem was difficult to account for in the initial design but was quite simple to implement as a modification of the first design.)

COMMANDS4: Update preliminary assumptions relating the Look and Find actions to the editor.

OFFUTL: The package "Offutl."

EDIT1: Modifications to the editor for forms and field handling.

LIST: The package "Offlst."

The resulting prototype program consisted of about one thousand lines (pretty-printed). The estimated development time of this program is about three or four weeks, although the actual elapsed time was longer because it included parallel development and debugging of some of the features of Castor and the underlying Ada system.

One feature of the Castor/Ada system which interfered with the development somewhat was the strict prohibition of forward procedure references. To reduce unnecessary writing, Castor does not require package specifications (although in pure Ada a package specification must be given for every package body). In Castor, a package body can be written by itself (as a subprogram body can), and in practice this makes the expression of the program much briefer. As a consequence, however, a certain amount of thought is required when assigning definitions to packages and when ordering the packages, in order to make sure there are no forward references.

USE OF MACROS IN THE PROTOTYPE

In concluding this demonstration of Castor, we give an example of the use of macros in the prototype. The calling forms

```
L: List Of (Datanode);
```

and

```
For (X) In (L) Loop
   BEGIN
      ...
   END;
```

and

```
Append1 (N) To (F.Filval);
```

are part of what might be a general list processing library package. Their definitions are as follows:

```
MACRO List Of (Element_ty) RETURN Namenode IS
  Node_ty: Namenode := New_id (Element_ty, "node");
  List_ty: Namenode := New_id (Element_ty, "list");
BEGIN
  IF Test_decl (Node_ty) THEN
     Add_decl (Gendecl ("TYPE $Node_ty;"));
     Add_decl (Gendecl ("" &
         "TYPE $List_ty IS ACCESS $Node_ty"));
     Add_decl (Gendecl ("" &
         "TYPE $Node_ty IS                " &
         "RECORD                          " &
         "    Nxt: $List_ty;              " &
         "    Val: $Element_ty;           " &
         "END RECORD;                "));
     Set_tag (List_ty, 1, Element_ty);
     Set_tag (List_ty, 2, Node_ty);
  END IF;
```

```
  RETURN List_ty;
END List Of;


MACRO For (X) In (L) Loop (S: ARRAY) RETURN Stmtnode IS
  List_ty: Intnode := Type_of (L);
  Element_ty: Intnode := Tag (List_ty, 1);
BEGIN
  RETURN Genstmt ("           " &
      "DECLARE                     " &
      "   Tmp: $List_ty := $L;     " &
      "   $X: $Element_ty;         " &
      "BEGIN                       " &
      "   WHILE Tmp /= NULL LOOP   " &
      "       $X := Tmp.Val;       " &
      "       $$S;                 " &
      "       Tmp := Tmp.Nxt;      " &
      "   END LOOP;                " &
      "END;                 ");
END For In Loop;


MACRO Append1 (E) To (L) RETURN Stmtnode IS
  List_ty: Intnode := Type_of (L);
  Node_ty: Intnode := Tag (List_ty, 2);
BEGIN
  RETURN Genstmt ("           " &
      "DECLARE                                    " &
      "   Tmp: $List_ty := $L;                    " &
      "   Ptr: $List_ty := NEW $Node_ty;          " &
      "BEGIN                                      " &
      "   Ptr.Nxt := NULL;                        " &
      "   Ptr.Val := $E;                          " &
      "   IF $L = NULL THEN                       " &
      "       $L := Ptr;                          " &
      "   ELSE                                    " &
      "       WHILE Tmp.Nxt /= NULL LOOP          " &
      "           Tmp := Tmp.Nxt;                 " &
      "       END LOOP;                           " &
      "       Tmp.Nxt := Ptr;                     " &
      "   END IF;                                 " &
      "END;                 ");
END Append1 To;
```

The subprograms used in these macro definitions are described in Chapter 5.

The following protocol shows two procedures which use these macros. In this protocol they are displayed, then executed, and then displayed in their expanded form. Note that the type declarations produced are not shown in the final print-out, since the enclosing declarative part is not shown here. The calling form "List Of (-)" expands into the type identifier "Integer_list," the calling form "Appendl (-) To (-)" expands into a "begin" block with appropriate declarations, and the "For (-) In (-) Loop BEGIN ... END" calling form expands into another "begin" block with appropriate declarations and containing a "while" loop.

```
$pp("ldemo");

PACKAGE BODY Ldemo IS
  L: List Of (Integer);

  PROCEDURE Init IS
  BEGIN
     L := NULL;
     FOR I IN 1 .. 5 LOOP
        Appendl (101 * I) To (L);
     END LOOP;
  END Init;


  PROCEDURE Print IS
  BEGIN
     For (X) In (L) Loop
        BEGIN
           Put (X);
           New_line;
        END;
  END Print;

END;

$init;
```

```
$print;
101
202
303
404
505

$pp("ldemo");

PACKAGE BODY Ldemo IS
  L: Integer_list;

  PROCEDURE Init IS
  BEGIN
     L := NULL;
     FOR I IN 1 .. 5 LOOP
        DECLARE
           Tmp: Integer_list := L;
           Ptr: Integer_list := NEW Integer_node;
        BEGIN
           Ptr.Nxt := NULL;
           Ptr.Val := 101 * I;
           IF L = NULL THEN
              L := Ptr;
           ELSE
              WHILE Tmp.Nxt /= NULL LOOP
                 Tmp := Tmp.Nxt;
              END LOOP;
              Tmp.Nxt := Ptr;
           END IF;
        END;
     END LOOP;
  END Init;


  PROCEDURE Print IS
  BEGIN
     DECLARE
        Tmp: Integer_list := L;
        X: Integer;
     BEGIN
        WHILE Tmp /= NULL LOOP
           X := Tmp.Val;
           Put (X);
           New_line;
           Tmp := Tmp.Nxt;
        END LOOP;
     END;
  END Print;

END;
```

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: TR-187	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Rapid Software Prototyping		5. TYPE OF REPORT & PERIOD COVERED: Technical Report
		6. PERFORMING ORG. REPORT NUMBER: Technical Report #187
7. AUTHOR(s): David Andrew Smith		8. CONTRACT OR GRANT NUMBER(s): MDA-903-82-C-0039
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Programming Environment Project, Computer Science Department, University of California, Irvine, California 92717		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Project Agency, 1400 Wilson Blvd, Arlington, Virgina 22209		12. REPORT DATE: May 12, 1982
		13. NUMBER OF PAGES: 184
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Western Regional Office, 1030 E. Green Street, Pasadena, California 91106		15. SECURITY CLASS. (of this report): UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Distribution is unlimited

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Rapid Prototyping	Ada
Programming Environments	Software Lifecycle
Programming Design Language	Stepwise refinement

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Rapid Prototyping is an approach to software development which emphasizes quick implementation of a working program. This dissertation makes two principal contributions. First, it provides concepts, techniques, and a philosophy of Rapid Software Prototyping and characterizes the benefits and limitations of its use. Second, it makes a contribution to programming environments which support Rapid Prototyping. An experimental language, Castor, is described

DD FORM 1473, 1 JAN 73 — EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

which was implemented to validate this approach in the prototyping of Ada programs*. The following summarize the main results of this research:

1. A statement of the purpose and value of Rapid Prototyping: Rapid Prototyping provides accelerated feedback to the early stages of analysis in the software lifecycle. This can be of great benefit when there are areas of risk that only experience with a working system can resolve.

2. A statement of the limitations of Rapid Prototyping: Rapid Prototyping cannot show the behavior of the final system in all respects. Careful planning is therefore necessary to determine the objective of the prototype and what sacrifices can be made in areas of low risk.

3. Techniques for Rapid Prototype Programming: Castor is both a Program Design Language (PDL) and an implementation language. The PDL nature of Castor arises from the use of free form descriptions called "calling forms." An agenda of undefined calling forms is provided interactively. Contributions in this area are that:

   a) Castor implements a refinement paradigm for the new language, Ada;

   b) Castor macro facilities are easy to learn and remember; and

   c) the Castor macro language is independent of the underlying program representation.

4. A stock of ideas for an "Ada laundry": An Ada laundry allows the user to relax temporarily the rules of pure Ada. This helps compensate for aspects of Ada which orient it more toward long program life than short term ease of expression.

Castor was used to build a prototype of moderate size which is described in an appendix.

---

*) Ada is a trademark of the United States Department of Defense.