UC Irvine

ICS Technical Reports

Title
Persistent prototypes

Permalink
https://escholarship.org/uc/item/3s25k7bw

Author
Willson, Stephen Hunter

Publication Date
1986

Peer reviewed

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

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Z
699
C3
no. 86-20

UNIVERSITY OF CALIFORNIA
Irvine

Persistent Prototypes

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

By

Stephen Hunter Willson

Committee in charge:
Professor Richard N. Taylor, Chair
Professor Nancy G. Leveson
Professor Rami R. Razouk

Technical Report 86-20

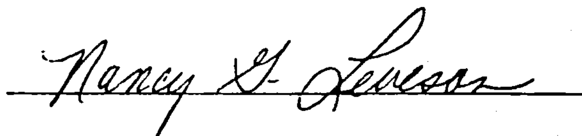
1986

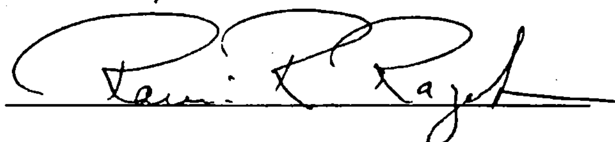
© 1986

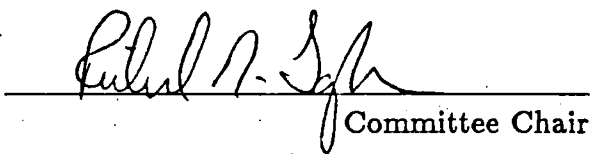
Stephen Hunter Willson

All Rights Reserved

The dissertation of Stephen Hunter Willson is approved,
and is acceptable in quality and form for
publication on microfilm:







Committee Chair

University of California, Irvine

1986

This dissertation is dedicated to the memory of
Walt Disney
whose commitment to quality
is a continuing source of inspiration
and to my wife
Kayler M. Clarke
who has helped me so much in so many ways.

Contents

List of Tables vi
List of Figures vii
Acknowledgements viii
Curriculum Vitae ix
Abstract xi

1 Research Objectives **1**
1.1 Introduction 1
1.2 Prototyping 3
1.3 Requirements of Production Environments 6
1.4 Prototype Promotion 7
1.5 Interactive Ada 9
1.6 Summary of Results 13

2 Rapid Prototyping Environments **15**
2.1 Interlisp 16
2.2 Cedar 20
2.3 Cornell Program Synthesizer 23
2.4 Pecan, Magpie, and Dice 25
2.5 C and Lint 27
2.6 Arcturus 29
2.7 Interactive Ada in Surveyor 31
2.8 Summary 37

3 Collecting Dependency Information **39**
3.1 Object Dependencies 40
3.2 Symbols 44
3.3 Type Descriptors 46
3.4 Internal Form 48

3.5 Mapping Functions 53
3.6 The Collection Process 54
3.7 Example 56

4 The Export Tool **63**
4.1 Algorithm TGEN 66
4.2 Algorithm TSORT 69
4.3 Algorithm BREAKCYCLE 71
4.4 Algorithm TESTCYCLE 75
4.5 Examples 77

5 Recapitulation **84**
5.1 Lessons Learned 84
5.2 Future Work 88
5.3 Summary 90

References **92**

List of Tables

2.1 Directory of Tektronix 4129 Package 38

3.1 Type Descriptor Components in Surveyor 47
3.2 List of Symbols 59
3.3 Dependencies from Type Descriptors 60
3.4 Initialization and Constraint Dependencies 60
3.5 Dependencies in lookup and enter 61
3.6 Summary of Dependencies 61

4.1 Summary of Dependencies 79
4.2 Predecessor Counts and Successor Sets 79
4.3 Results of BREAKCYCLE 83

List of Figures

3.1 Layout of Surveyor Workspace 42
3.2 Workspace After Redefining ca 43
3.3 A Symbol 44
3.4 Sample Type Descriptor 49
3.5 Factorial Function 51
3.6 BIF Tree for Factorial Function 52
3.7 Dependency Graph 62

4.1 Ordered Interactive Ada Fragments 78
4.2 Sorted Dependency Graph 81
4.3 Sample Cyclic Graph 82

ACKNOWLEDGEMENTS

Professor Richard N. Taylor provided many helpful suggestions regarding the structure of my dissertation.

Thanks to Professors Nancy G. Leveson and Rami R. Razouk for serving on my committee and providing helpful advice.

Professor Thomas A. Standish introduced me to the fields of programming environments and "rapid prototyping."

Professor John L. King helped me focus my attention on defining the problem I wanted to solve.

The Surveyor Ada interpreter is based on the Arcturus interpreter developed by the Programming Environment Project. Professor Standish was responsible for obtaining our funding and for building a working relationship with the Software Productivity Project (SPP) at TRW. Steve Whitehill and I wrote the first version of the interpreter in C [27] including the static scoping and type structure. Steve also wrote the code for unconstrained array types and aggregates. Frank Tadman added several control constructs including exception handling. Frank also replaced our hand-written parser and lexer with one produced by *yacc* [25] and *lex* [30]. Craig Snider maintained Arcturus for several years. Scott Auchmoody wrote a version of put which prints the values of arbitrarily complex data types. Ray Klefstad wrote the code for case statements.

Several members of the Software Productivity Project at TRW provided help. Barry W. Boehm provided insight on the need for evolving software from prototype implementations. Frank Belz and Judy Bamberger used early versions of Arcturus. Their experience with the system helped me identify the need for an export capability.

Kayler M. Clarke helped with algorithm BREAKCYCLE. Kayler also provided intellectual, emotional, and spiritual support during the last two years of the work. In particular, Kayler acted as my own "export laundry," helping me turn my stream of consciousness style into something more structured. The many hours we spent discussing the nature and goals of scientific research were highly beneficial. Without her help I would not have completed this dissertation.

I am indebted to my friend Rama for teaching me various concentration, meditation and survival techniques, which have aided me greatly not only as a graduate student but in many areas of life.

I appreciate the financial support of my parents, Richard and Beverly Willson, and the Northrop Corporation.

Many thanks to Jim Reis, Manager of the Autonomous Systems Laboratory at the Northrop Research and Technology Center (NRTC) in Palos Verdes,

California, for encouraging me to complete my dissertation. Jim gave me the time and resources I needed to complete the work.

Thanks to Marty Cohen for showing me reference [19], and lending me [4] and [9]. Also, Joan Willis, the NRTC librarian, helped me track down several references.

This dissertation was computer typeset using the LaTeX macros by Leslie Lamport; TPIC, by Brian W. Kernighan and extended by Tim Morgan; and the TeX typesetting system by Donald Knuth. Various computer resources at UCI and NRTC were utilized to typeset the text: the NRTC Vax-8600 (Venus), the NRTC Computer Science Lab's ISC-68020 (Gremlin), one of UCI's Vax-750s (ICSC), the UCI Sequent (Bonnie), two of UCI's Sun-3s (ICSG and ICSH), two of UCI's Sun-2s (Sun1 and Sun2), and an earlier draft on Kayler Clarke's IBM-PC (Rum Punch); and various laser printers at both sites. Thanks also to Marshall Rose for providing me with great quantities of information that aided me in the preparation of this manuscript. Marshall also provided access to Gremlin and was instrumental in installing the Internet connections at both UCI and NRTC.

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.

The table below lists various trademarks referenced herein and the organizations to whom they are registered.

Organization	Trademark(s)
American Mathematical Society	TeX
AT&T	*Unix*
Digital Equipment Corporation	*VMS, Vax, Vax-8600*
Integrated Solutions, an NBI Company	*ISC-68020*
International Business Machines Corp.	*IBM, IBM-PC*
Sequent ??	*Sequent*
Sun Microsystems Inc.	*Sun-2, Sun-3*
Tektronix, Inc.	*Tektronix*
United States Government (Ada Joint Program Office)	*Ada*

CURRICULUM VITAE

Stephen Hunter Willson

August 23, 1959	Born Los Angeles, California
1977	National Merit Scholarship Finalist Corona del Mar High School
1981	B.S. in Information and Computer Science (CUM LAUDE) University of California, Irvine
1982	M.S. in Information and Computer Science University of California, Irvine
1986	Ph.D. in Information and Computer Science University of California, Irvine

ABSTRACT OF THE DISSERTATION

Persistent Prototypes

by
Stephen Hunter Willson

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1986

Professor Richard N. Taylor, Chair

Rapid prototyping environments (RPEs) give the programmer great flexibility during the composition phase of program-creation. An RPE enables the designer/programmer to define and redefine fragments of programs in an ad hoc fashion with a minimum number of constraints.

Existing RPEs are based on programming languages which are unsuitable for use in long term production environments. This inability to move from the ad hoc development environment to a more structured production environment inhibits the long term value of the prototype: most prototypes are thrown away and the work is recreated in a compiler based environment. A *persistent prototype* is defined as a prototype which evolves into a mature program. Persistent prototypes are valuable because they eliminate the need to recreate work done in the prototyping environment in a compiler based environment.

This dissertation describes the requirements for rapid prototyping environments based on languages with strong typing, which are suitable for production use. Prototypes are written in interactive Ada, a variant of standard compiled Ada, which is suitable for prototyping. Interactive Ada relaxes the compiled Ada constraint that definitions are stored in a particular order. Thus, it is necessary to *export* the prototype from the interactive environment to a compiled environment by sorting the declarations into an appropriate order. The dependency relationships inherent in Ada's strong typing make this sorting process possible. The dissertation details the data structures and algorithms necessary to effect this transfer.

Chapter 1

Research Objectives

> **persist** *v.* **1.** To hold firmly and steadfastly to a purpose or undertaking despite obstacles. **2.** To continue in existence. [Lat *persistere.*]
> *The American Heritage Dictionary* (1983)

This chapter presents an overview of the research objectives and principle contributions.

1.1 Introduction

Rapid prototyping environments (RPEs) give the programmer great flexibility during the composition phase of program-creation. An RPE enables a designer/programmer to define and *redefine* fragments of programs in an ad hoc fashion with a minimum number of constraints.

Existing RPEs are based on programming languages which are unsuitable for large scale development. This inability to move from the ad hoc development environment to a more structured development environment inhibits the long term value of the prototype: most prototypes are thrown away and the work is recreated in a compiler based environment. A *persistent prototype* is defined as a prototype which evolves into a mature program. Persistent prototypes are valuable because they eliminate the need to recreate work done in the prototyping environment in a compiler based environment.

This dissertation describes the requirements for rapid prototyping environments based on languages with strong typing, which are suitable for production use. Prototypes are written in interactive Ada, a variant of standard compiled Ada, which is suitable for prototyping. Interactive Ada relaxes the compiled Ada constraint that definitions are stored in a particular order. Thus, it is necessary to *export* the prototype from the interactive environment to a compiled environment by sorting the declarations into an appropriate order. The dependency relationships inherent in Ada's strong typing make this sorting process possible. The dissertation details the data structures and algorithms necessary to effect this transfer.

1.2 Prototyping

Sometimes it is difficult to accurately and completely specify the requirements for a particular software system [8, page 290]. One technique used to experimentally determine software system requirements is *prototyping.* Prototyping is a methodology for building a subset of a system and testing portions of the behavior with potential users to determine if the design meets their requirements.

> "In most fields of engineering the significant new project costs are labor, materials, and capitol [sic]. In software engineering, material costs are small and capitol [sic] costs are often fixed. Labor, and therefore time, is the main project expense. Scale modeling of software, if it is to be cheaper than building the whole system, must take the form of quickly built models. Thus the term 'rapid prototyping'." [50, Page 181]

There are generally two approaches to rapid prototyping.

> "Discussions of rapid prototyping generally focus on two strategies for producing the prototype: methodologies and executable specifications. The methodology strategy, in its simplest form, suggests that the prototype be written in a language which facilitates expression and experimentation, possibly at the cost of efficiency or robustness. It is a strategy that we all already use (or should use if we don't), at least for small prototypes. The executable specifications strategy involves describing the software to be prototyped in some (relatively formal) specification language which has some operational semantics so that it can be directly executed (or interpreted) in some fashion." [5, page 33]

The focus here is on the methodology approach rather than the executable specifications approach. As indicated above, a prototype is often developed in

a *rapid prototyping language* (RPL). Rapid prototyping languages typically are interpreted, lack strong typing, and have minimal structural requirements. Lisp, Basic, Forth, and APL are languages commonly used for rapid prototyping. A rapid prototyping language is used in conjunction with a *rapid prototyping environment* (RPE). The RPE provides useful services which extend the power of the RPL. "Using environments, prototypes are used to clarify requirements and design issues prior to construction of the actual product." [42, page 2].

Normally when developing a prototype in an RPL, one makes great use of the services provided by the runtime (i.e., interpreted) environment. "One of the valuable lessons we learned from Interlisp and Smalltalk was that the availability of an interpreter greatly facilitates debugging and testing..."[46, page 286] For instance, the Lisp `(print (eval (read)))` loop is often modified to provide specialized input and output processing. Compile time type checking is not considered important and one relies on the interpreter to catch significant errors. The error recovery services of the interpreter are used instead of special case code (e.g., to check for a null pointer before dereferencing it).

Since the nature of the task is inherently experimental, it is bad policy to overconstrain one's options by hiding subroutines and types that might be useful at a later time. Thus, the rapid programming environment usually supports

some means (i.e., a database) for accessing a large number of separate functions and data structures.

The methodology of prototyping and the programming environment used to carry out that task are intimately connected. "The quality of software depends primarily on the programming methodology in use. The choice of programming language, however, can have a major impact on the effectiveness of the methodology." [31, page 565] Similarly, the choice of programming environment impacts the effectiveness of the methodology as well. "...every detailed recommendation on how to write programs is also a recommendation on how to design an interactive programming system that supports the methodology." [38, page 37]

The characteristics of a rapid prototyping environment that supports the prototyping methodology are summarized below:

- The designer/programmer requires rapid system response to commands to execute, suspend, modify and debug a program.

- The designer/programmer requires fine grained interaction which allows him/her to experiment directly in a "free form" manner with data structures and routines, without waiting for a traditional compiler and or linker.

- The designer/programmer needs to enter definitions rapidly, without worrying too much about getting everything in the right order. The programmer shouldn't spend time positioning each new routine in the proper place relative to the others.

- The designer/programmer needs access to a potentially large set of data structures and routines. The environment needs to support a large, easy to manipulate database of these definitions.

- The designer/programmer should not be overconstrained by environment and/or language rules for type checking. The designer/programmer interactively defines new types and alters old ones. The key characteristic here is the ability to *redefine the type of an object.* This capability allows the designer/programmer to quickly implement major structural changes to the data types defined in the prototype.

1.3 Requirements of Production Environments

The requirements for production quality programs, namely, reliability and maintainability, have resulted in a development methodology that, generally speaking, is in conflict with the prototyping methodology. The following list, paraphrased

from [21, pp. 202–203], describes the characteristics of production quality software:

- The system has to function correctly. Even small errors can be costly.
- The system is long-lived. The cost of developing the system is recouped over a long period of time.
- During its lifetime, the system undergoes considerable modification.
- Many people—tens or hundreds—are involved in the development of the system.

The goals of a production environment are to promote stability, reliability, and maintainability. These goals conflict directly with the goals of a prototyping environment, namely, to promote flexibility and experimentation. Interactive development environments tailored to the needs of one or two designer/programmers do not adequately support the needs of large teams of programmers and will not do so in the foreseeable future.

1.4 Prototype Promotion

The differences in the two kinds of program development just described have led to the notion that prototype programs are significantly different from production

programs. But, as mentioned in section 1.2, labor is the most significant cost of developing software. To avoid wasted effort (and thereby reduce costs and increase productivity) it is desirable to reuse as much of the prototype software as possible when constructing the production version of the same system. This dissertation proposes the following strategy for reconciling the two methodologies in such a way that it is possible to evolve a prototype implementation into a production quality implementation.

1. Develop the prototype in a programming environment that supports *interactive prototyping in Ada.* This environment supports the kind of interaction normally expected in a rapid prototyping environment, but in the context of Ada, a strongly typed, statically scoped, modular language.

2. At the appropriate time, export the prototype from the RPE to a traditional, compiler based programming environment.

3. Continue development of the program in the traditional environment, using the traditional tools of large software development.

For this approach to work, an RPE for Ada must exist. Since the RPE treats Ada as if it were a prototyping language, it may be necessary to repackage the prototype into a more standard form before continuing development in the

compiler based environment.

The next section introduces the key concepts of *interactive Ada*, and indicates how interactive Ada is used both as a prototyping language and a production quality language. There are minor differences between interactive Ada and standard compiled Ada. These minor differences have a minimal impact on the semantics of the language, but have a large impact on the nature of the programming environment used to develop Ada programs.

1.5 Interactive Ada

Surveyor [52] is a programming environment which supports interactive input and execution of *fragments* of Ada programs, including "on-the-fly" redefinition of types, variables, and subroutines. Interactive Ada, as implemented in Surveyor, is fairly similar to compiled Ada. There are four principle differences between interactive Ada and compiled Ada:

1. Type checking occurs at runtime.

2. A program fragment (e.g., a single statements or declaration) is the basic unit of interaction (rather than an entire compilation unit).[1]

[1]The valid top level inputs to the Ada interpreter are as follows: expression, simple statement, unlabeled compound statement, use clause, object declaration,

3. Programs are not stored in any particular "order." Since it is possible to enter an individual declaration without placing it in the context of all other declarations as required in compiled Ada, it is necessary for the programming environment to take on the burden of connecting together all declarations at runtime.

4. It is possible to *redefine* the type of an object. Again, since the designer/programmer can enter individual declarations interactively, it is possible for him to redefine an existing declaration.

Taken together, these differences significantly alter the way the designer/programmer interacts with the system. The compile-link-debug cycle is eliminated. These alterations to the normal use of compiled Ada make interactive Ada suitable as a prototyping language without sacrificing the needs of a production environment (e.g., strong typing and modular structure).

Ada as an RPL

Surveyor's interactive Ada meets the requirements of a prototyping environment:

---

type declaration, subprogram declaration, task declaration, rename declaration, number declaration, subtype declaration, package declaration, exception declaration, subprogram body, task body, package body, and body stub.

- Programs are interpreted. Modifications to subroutines or other definitions are instantly available for testing.

- A single statement, declaration, or expression is the level of interaction with the interpreter.

- Programs are not stored in any particular order. The environment takes care of connecting definitions with usages at runtime.

- Definitions are usually stored in one flat scope. The normal Ada semantics which allow nesting of modules is intact, but for prototyping purposes the designer/programmer usually stores everything in one scope.

- Type checking occurs at runtime. Even though the interactive Ada interpreter enforces strong typing, the ability to *redefine the type of an object* is still available to the designer/programmer. In fact, type definitions themselves can be redefined, thus causing global changes to the structure of a prototype. Strong typing encourages good software practice by helping the designer/programmer build hierarchical definitions for complex types. The ability to redefine types makes interactive Ada suitable as a prototyping language without sacrificing strong typing.

Ada as a production language

The differences between interactive Ada and compiled Ada do not interfere with its use in a production environment.

- Once an interactive Ada program is translated to compiled Ada, the global type checking facilities of the compiler are used in the normal fashion.
- Again, once an interactive Ada program is translated to compiled Ada (*exported*), the runtime overhead of the development environment is completely eliminated.
- The facilities for modular program structure remain intact. Since compiled Ada and interactive Ada are so similar, the goals of distributed development among many individuals, production of maintainable code, and management control of module interfaces are nicely supported.

Once strong typing is introduced as a characteristic of an RPL, a mechanism is required to support the *export* of the prototype from the development environment to the production environment. This need arises because the RPE (Surveyor) does not store programs in any particular order, whereas a compiled Ada environment has strict rules defining a required order for the textual representation of a program [1, page 3-43]. An *export tool* provides the service of

packaging up interactive Ada programs into compilable versions of the same. The Ada interpreter records the dependency relationships between definitions in the RPE. The export tool uses these dependency relationships to sort the definitions into an appropriate order for outputting as a compilable file.

1.6 Summary of Results

This dissertation discusses a programming environment for building *persistent prototypes*. A persistent prototype is one that is designed to be kept as the basis for a more mature system. It is beneficial if the persistent prototype is developed in a language with strong typing and modular static scoping (e.g., Surveyor's interactive Ada). The quality of the prototype is improved (because of strong typing), and the prototype can be further developed and refined in a production (i.e., compiler based) environment. This approach differs from traditional techniques which use dynamically scoped, loosely typed languages for implementing prototype systems. Without strong typing and modular structure, traditional prototypes fail to satisfy the requirements of production environments, and therefore can not "scale up."

Chapter 2 presents a survey of related work in programming environments, with particular emphasis on the suitability of each environment for developing

persistent prototypes. Chapter 2 concludes with a description of Surveyor, and indicates how it is used to build prototype systems.

Chapter 3 explains how the dependency relationships between type definitions are stored in the Surveyor environment. The issues relating to storing programs as unordered fragments are explored, and the need for an *export tool* to effect the transfer from Surveyor to a more traditional environment is explained. A process for collecting these dependencies into tabular form is presented as well.

Chapter 4 presents several algorithms which work together to turn the table of dependencies generated by the process described in Chapter 3 into an ordered list of declarations suitable for use with a compiled environment. The algorithms deduce an ordering for the "unordered" definitions stored in Surveyor. This ordering serves two purposes:

1. The ordering illuminates the structure implicit in the dependency relationships between various type definitions, and

2. The ordering allows the definitions to be compiled with a standard compiler.

The ideas presented in this dissertation point the way to improved programming environments that support the need for both flexibility and imposed structure.

Chapter 2

Rapid Prototyping Environments

> Many of Fortran's restrictions, such as the number of array dimensions or the form of expressions used as array indices, are based directly on what could be implemented efficiently on the IBM 704.
>
> The issue of efficiency has changed considerably, however. Efficiency is no longer measured only by the execution speed and space. The effort required to produce a program or system initially and the effort required in maintenance can also be viewed as components of the efficiency measure.
>
> — Carlo Ghezzi and Mehdi Jazayeri,
> *Programming Language Concepts* (1982)

This chapter presents a survey of related research on programming environments. Many of today's research programming environments are strongly influenced by the Interlisp programming environment [45,38,47]. The design of Surveyor, in particular, is heavily influenced by Interlisp.

2.1 Interlisp

Interlisp is one of the most mature programming environments. It supports a dialect of Lisp that has been augmented by many environment-supported functions.[1] There are so many features in Interlisp that it is difficult to identify them all. The important features that influenced Surveyor are as follows:

1. Code and data are the same. Of course, all data in Lisp is either a list or an atom. Programs are stored using a list representation, so naturally, programs *are* data. This equivalencing makes it easy to write programs that create or modify other programs.

2. The programmer has easy access to internal data structures, including symbol tables (the OBLIST), and program state information (e.g,. backtrace).

3. New functions are easy to add to the repertoire of functions already in Interlisp. These are callable in the same way as built-in routines.

4. The user interacts with the interpreter at a fine grain of interaction.

[1]In this discussion the incremental compilation facilities included in Interlisp are ignored; while these facilities are important, they do not significantly alter the interaction between the user and his program. Functions are still invoked from the interactive interpreter and do not stand alone. The compilation facilities improve the execution speed of Lisp programs but depend heavily on the interpreter runtime environment.

5. The system maintains an extensive runtime state even though the user changes his programs. In other words, when the Interlisp programmer changes a function, the data structures and variables he has created are not destroyed by the system.

6. The debugging language is the same as the programming language.

7. The command language is the same as the programming language.

All of these capabilities are present in Surveyor, though sometimes in slightly different form.

1. Code and data are the same. Programs are represented as trees. The concrete representation of these trees is defined by standard Ada type definitions which are accessible to the user. These trees are manipulated in the same way all Ada data structures are.

2. The user has access to internal data structures. The runtime stack, symbol tables, and type descriptors are all available to the user for analysis.

3. New routines are added simply by defining them to the top level. They are invoked in the same was as all built-in routines, namely, by a procedure or function call.

4. The granularity of interaction is small; the user can enter a single statement, declaration, or expression.

5. The user declares variables and types to the interpreter and these retain their values and definitions throughout the entire session.[2]

6. Debugging is accomplished by invoking built-in Ada functions that alter the normal flow of control (i.e., the cause execution of the program to suspend and control passes to a *break package*). Once in the break package, built-in procedures are used to analyze the state of the program (e.g., examine variables or the call stack).

7. The command language is the same as the programming language. Commands for reading and writing files and otherwise altering the programming environment are given using standard Ada semantics.

Although the Interlisp environment is a very good programming environment, the fact that it is based on Lisp hinders its use for some kinds of applica-

[2]Here, a session is defined as the time beginning when the user starts the Surveyor interpreter to when he finally exits the system by invoking the procedure *quit.* The operating system is given the burden of providing a means of suspending a session and later resuming it. With the proper operating system support, the user need never terminate a particular session at all, but merely suspend and resume it. In this case, the lifetime of these user defined variables and types is indefinitely long.

tions.

1. Lisp environments are tailored to applications that manipulate lists. Working with more structured data types (e.g., multi-dimensional arrays of records) is difficult.

2. Lisp environments do not provide an adequate syntax for constructing abstractions for numerically-intensive applications (e.g., overloading of operators and encapsulation of abstract representations into packages).

3. Transferring an application from a Lisp environment to an embedded system is impossible: Lisp applications depend heavily on the Lisp programming environment and cannot run without it.

4. Dynamic scoping and loose typing are unreliable ways to structure a program [20]. Even supposing a Lisp prototype could be exported, the resulting code would not be suitable for use in a production environment.

Recent attempts at imposing static scoping on Lisp have met with some success. Scheme [22], T [2], and Common Lisp [40] are dialects of Lisp with static scoping. T, in particular, defined a variant of Lisp where the compiled and interpreted semantics are identical: both use static scoping and require predeclaration of special variables. In previous Lisps (as well as the more recent

Common Lisp), the compiled and interpreted semantics sometimes differed. In order to compile his/her code, the programmer annotated his/her already working program with type definitions for those variables that conflicted with the default data type (a dynamically scoped list). Once these definitions were introduced, they sometimes interfered with the loose typing and dynamic scoping rules of the interpreter. The only resolution to this problem thus far has been to impose static scoping throughout. This is the approach used in Surveyor, and implemented by adopting Ada semantics.[3]

2.2 Cedar

The Cedar environment [46] is similar to the Interlisp environment, except Cedar supports Mesa [33], a strongly typed, statically scoped language similar to Ada. "...A fair characterization of the Cedar project as it is currently constituted is that it is an attempt to take the Mesa language and build for it a programming environment based on ideas and techniques from Interlisp and Smalltalk." [46, page 286].

Both Cedar and Surveyor are based on the same requirements document

[3]In Surveyor, it is possible to dynamically bind a fragment of code into a specific scope at runtime; even so, once bound into place, the code obeys the normal static scoping rules.

[15]. This document outlined the characteristics of interactive environments as they were known at the time while including various requirements which at that time were only supported by traditional compiled environments.

Cedar and Surveyor are similar in many ways, but the scope of the Cedar project far exceeds that of Surveyor. Surveyor is focused on supporting interactive prototyping; Cedar includes document preparation, electronic mail, an advanced user interface, and shared (i.e., networked) file systems.

The relevant portion of Cedar for this discussion is the *UserExec*. The UserExec is a line-at-a-time interface to Cedar for Mesa programmers. "One of the valuable lessons we learned from Interlisp and Smalltalk was that the availability of an interpreter greatly facilitates debugging and testing..." [46, page 290]. The Cedar interpreter is used to call subroutines and evaluate expressions. The user can assign the results to temporary variables and then use those results in later expressions. The UserExec also accepts debugging commands (e.g., setting and clearing breakpoints).

The UserExec appears to work by loading a symbol table for each module being debugged. The module consists of compiled code which is connected to the source code by a table of pointers stored in the symbol table. The user can enter a text editor anytime to view the source code connected to the current

compiled module. Breakpoints are set by inserting a breakpoint instruction into the compiled code. The symbol tables also contain detailed type descriptors for each type used or defined in a module.

The UserExec is a powerful debugging tool. The expression interpreter allows flexible interrogation and alteration of the state of a program. Changes to the program are made by editing a module, and then recompiling. The new code is linked to the existing code. It is not possible to change the code while it is executing.

While Cedar and Surveyor have similar program construction and debugging facilities, there are several significant differences. Surveyor contains a more extensive interpreter: the user can enter or change type definitions, variables, procedures and functions; enter compound statements (e.g., loops and conditionals); as well as evaluate expressions.

The granularity of incremental program changes is much larger in Cedar: the smallest unit of change is the module, rather than an individual statement or declaration, as in Surveyor. As a result, the burden of keeping a program in order still falls on the programmer.

2.3 Cornell Program Synthesizer

The Cornell Program Synthesizer (CPS) [48] is programming environment for interactive input and execution of PL/C programs. CPS focuses are incremental checking of user programs. Not only is the program checked and pseudo-code generated in the environment, but the code is executed there as well. Additionally, incomplete programs can be executed; when the interpreter finds a "hole" in a program, it stops and alerts the user. The user can modify his program with the syntax-directed editor front-end, and then restart it. Obviously, programs created with CPS are exportable. One merely takes the program text and compiles it with an external compiler.

The Cornell Program Synthesizer is similar to other syntax-directed systems (e.g,. POE [17], ALOE [32], and Mentor [16]), in that it focuses on incremental changes to a single monolithic program. This differs significantly from the approach in **Surveyor**, where the user enters fragments of programs stand-alone (i.e., without having to explicitly modify the structure of the entire program).

The Synthesizer Generator is a tool developed by Thomas Reps for generating CPS-like environments [36,34]. The most important contribution of this more recent work is the use of attribute grammars to specify semantic transformations as well as the syntactic transformations. The use of an attribute grammar for

describing the editor opened up the possibility of formalizing the update process, which Reps did in his dissertation [34], which won the ACM Doctoral Dissertation Award in 1983. By formally specifying the semantics of the language, it is possible to automatically generate a structure editor that checks for semantic inconsistencies. The major drawback with this approach, in terms of rapid prototyping, is that the feedback loop is too tight: deleting the definition of a single variable or type definition may cause large portions of the resulting program to become invalid. Changing the type of a variable may have an equally large impact. This makes small-grained experimental changes difficult.

In Surveyor, the consistency of the program structure is only checked at export time or at runtime. As a result, the designer/programmer is free to experiment with temporarily invalid constructs, and even to execute these constructs without invalidating his previous work. Unfortunately, the Synthesizer Generator does not automatically create interpreters for languages specified with the attribute grammar formalism. Adding this capability, as well as considering a mechanism for delaying consistency checking, would increase the value of the Synthesizer-created editors for rapid prototyping purposes. Delayed type checking and interpretation of incomplete programs are powerful techniques for experimental prototyping.

2.4 Pecan, Magpie, and Dice

The following three systems are important because each in a different way promotes experimental checkout of portions of a working system at runtime.

Pecan is another structure editor based system [35]. In this particular case, though, the focus is on displaying different simultaneous semantic views of an executing program. For rapid prototyping purposes, the most important feature of Pecan (which implements a Pascal [23] interpreter), is the incremental semantics that allow the program to be compiled into pseudo-code *as it is being edited.* This feature provides the same kind of support for Pascal which Surveyor provides for Ada and Interlisp for Lisp, namely, modification of a program without destroying the existing global data already created by the program's execution.

Magpie [14] is an interactive Pascal environment which also features incremental compilation, and therefore avoids the need for a debugging language (since the user simply modifies the code directly [inserting/removing print statements, etc.]) The focus of the research was on making the programmer unaware of the various modes the system is in (e.g, compiling, editing, etc., appear as one uniform activity), The Magpie system has a particular feature that is beneficial for rapid prototyping. Magpie allows immediate mode execution of statements, including assignment; therefore the user can build data structures by hand. The

programmer does this by entering assignment statements or calling already defined procedures. The user enters these statements in a special browsing window called the *workspace*. The workspace defines an anonymous procedure which contains the statements the user wants to execute. The user does this when he/she wants to quickly checkout the effect of calling certain procedures. Magpie does not allow local definition of variables or types within the workspace.

Another system with the same important capability for modifying a program while it is executing is Dice [18]. Dice, or Distributed Incremental Compiler Environment, supports incremental compilation of Pascal code on a remote target computer. Again, the relevant capability is modification of the source code while the program is executing. The programmer can insert print statements directly into code that has been compiled and loaded on a remote target. In addition to the benefit that the programmer does not need to learn a special debugging language, the global program state remains intact even though portions of the code have been modified.

The main problem with these systems, from a rapid prototyping point of view, is that the programmer is still constrained to get everything in the right order. Each system will *check* for consistent declarations and usages, but does

not provide any *assistance* with the problem of organizing the code.[4] Dice, in particular, maintains complete cross reference information in order to implement the incremental update of the compiled code on the remote target: this cross reference information is exactly the information required for automatic ordering of the declarations within a scope. The power of the DICE system as well as the Pecan system, for rapidly prototyping systems, would be increased with the addition of automatic program ordering.

2.5 C and Lint

Rapid prototyping is not always done with interactive environments. Many prototype systems are developed in C [27]. The language is flexible enough such that the programmer can make major structural changes without a major rewrite. Because the language is independent of the development environment, exporting applications is not a problem. Many prototype C applications evolve quickly into production versions [37].

The development environment is a loosely coupled collection of tools: an editor, a compiler, a linker, and hopefully, a debugger. Other tools are some-

[4]This is not exactly true: The Pecan system provides a special editor which makes manually moving declarations around easier to do.

times present but this is the basic set. The Unix [26] operating system and its many tools is a programming environment tailored to C; simpler C programming environments exist also on a wide range of machines, from IBM-PCs to Vax/VMS.

The key aspect of C for rapid prototyping is default argument types and function return types. Coupled with the loose type-checking rules, it is easy to add new procedures and variables without significant concern for where they are declared. Of course, in the case of C, the same flexibility that is beneficial in prototyping interferes with creating reliable systems. The loose type checking rules allow inconsistencies to creep into programs undetected.

It is interesting to compare *lint* [24], the Unix tool for increasing the level of type checking in C programs, with the notion presented here of an export tool. In some sense, the C designer/programmer develops a prototype by ignoring some kinds of type checking. He then exports the prototype for use in a more stable, less flexible environment by analyzing the code with *lint* and fixing any problems. These fixups are performed manually even though the detection of the problems is done by *lint*. Unfortunately, many C programs are never checked by *lint*.

Standard C programming environments do not support line-at-a-time execution of C language constructs, but a new breed of interpreter based C envi-

ronments is emerging for use on personal computers [19]. They do not presently support line-at-a-time execution of C language constructs but it is not difficult to imagine this capability becoming available in the future.

2.6 Arcturus

Arcturus is the name of a "blue-sky" programming environment broadly defined by Thomas A. Standish. The design of Arcturus has, at one time or another, encompassed advanced user interfaces (e.g., lining the walls one's office with flat panel displays); novel input devices (e.g., a chair like Captain Kirk's which is used to fly around one's files); an integrated *Unit Development Folder* (UDF) [7, pp. 607–612] database; a program transformation system like the one in ECL [12]; a program design language [39,44]; a "template assisted editor" [28]; an interactive Ada interpreter [51]; and other interesting ideas. Many of the ideas were, of course, never realized. The Arcturus system as distributed by the Irvine Programming Environment Project consisted of the Ada interpreter and various tools, including the uniform user interface, the prototyping language, a fancy prettyprinter, and other tools designed and implemented by the graduate students in the Project.

Of particular interest here is the interactive Ada interpreter designed by

Willson, Tadman and Whitehill [51]. The requirements for the system were based loosely on the environment capabilities list in [15]. In fact, only the Ada interpreter and a simplified prettyprinter survive in Surveyor, as the result of an increased focus of attention by the author on persistent prototyping. In addition, the decreased complexity of the system made it easier to change as the research progressed.

There are several important enhancements in the Surveyor interpreter not present in Arcturus:

1. Single stepping and tracing of statements;
2. Access to internal data structures;
3. Modified internal type descriptors; and
4. The addition of the export tool.

Of these, the last two are the most significant. The type descriptors in Arcturus failed to retain the *fully qualified name* of the type as part of its description. This deficiency made rederiving a valid ordering nearly impossible, because the dependency graph was not directly accessible. The names were not retained because one of the goals of the Arcturus interpreter was to perform type checking early on at the expense of flexibility: "Unfortunately, at the moment

much type checking is performed at run-time, though it properly belongs in the interpreter's front-end." [41, page 59].

Of course, since Arcturus was going to perform so much checking in the front-end to the interpreter, it would still be necessary to get all of the declarations within a particular subprogram or package in the right order. In this case, as with the other systems described previously, there is no need for an export tool.

2.7 Interactive Ada in Surveyor

The Surveyor programming environment is a successor to Arcturus. The main changes, as summarized in the preceding section, are a modification to the representation of type descriptors and the addition of the export tool. In general, there is a change of focus from Arcturus' proposed early binding of usages to definitions, to binding at runtime.[5]

Slight modifications to compiled Ada semantics make Surveyor's interactive Ada suitable for prototyping. As stated in section 1.5, interactive Ada programs

[5]The typical style of interaction with Arcturus was to define everything within the context of one large procedure: in this way, the order of definitions was maintained explicitly by the author of the prototype. This style of interaction was necessary because there was no other way to organize the definition of complex data structures. In this respect, Arcturus offers little improvement over the NYU Ada interpreter [13]. The NYU interpreter also executes monolithic program structures without any sort of dynamic reordering of declarations ala Surveyor.

differ from compiled Ada programs in four principle ways:

1. Type checking occurs at runtime.

2. A program fragment (e.g., a single statement or declaration) is the basic unit of interaction (rather than an entire compilation unit).

3. Programs are not stored in any particular "order."

4. It is possible to *redefine* the type of an object.

These four principles make interactive Ada suitable for prototyping. For readers familiar with compiler-based environments, the last two principles are the most interesting aspects of interactive Ada.

Compiled Ada programs conform to a defined *order of elaboration*, where every symbol is *introduced* before it is used. This is true in interactive Ada as well. Objects must be defined before they are referenced or an error occurs. This implies that the definitions are entered in a certain meaningful order, however, *the initial user-specified ordering disappears as new definitions replace old ones.* A correct ordering is nevertheless deducible from the dependency information inherent in strong typing.

The name Surveyor comes from an analogy to cartography. The job of the cartographer is to generate a printed two-dimensional representation (a map) of a

physical toplogy. Analogously, Surveyor analyzes the topology of an Ada program and generates a one-dimensional representation (a listing) of that topology.

Examples of Interactive Ada in Surveyor

This section illustrates the prototyping capabilities of interactive Ada in Surveyor.

The example illustrates a small set of routines for manipulating a Tektronix 4010 display. Below, move and draw are primitive routines for moving the beam position and drawing from the current beam position to another point. These routines translate the given point into a special Tektronix encoding.

The plus sign ("+") is a continuation prompt from the system indicating it is waiting for the end of a syntactic unit (statement, declaration, or expression).

```
Surveyor> procedure move(p : point)
+ is
+ begin
+     code("LF");
+     put_point(p);
+ end;
Surveyor> procedure draw(p : point)
+ is
+ begin
+     code("LG");
+     put_point(p);
+ end;
```

The user next specifies a `point` as a series of three coordinates (x, y, and z).

```
Surveyor> type coordinate is (x, y, z);
Surveyor> type point is
+              array(coordinate range x..z) of integer;
```

Finally, the user enters the definition for the routine put_point.

```
Surveyor> procedure put_point(p : point)
+ is
+    hix, lox, hiy, loy, extra : integer;
+ begin
+    hiy := p(y) / 128 mod 32;
+    extra := p(x) mod 4 + (p(y) mod 4) * 4;
+    loy := p(y) / 4 mod 32;
+    hix := (p(x) / 128) mod 32;
+    lox := p(x) / 4 mod 32;
+    outchar(hiy + 32);
+    outchar(extra + 96);
+    outchar(loy + 96);
+    outchar(hix + 32);
+    outchar(lox + 64);
+ end;
Surveyor>
```

Both move and draw call the routine put_point which translates a point into the appropriate Tektronix escape codes. Notice the definition of point is postponed until *after* move and draw are defined. This is perfectly legal in interactive Ada because the local definitions (including the paramaters) in the scopes of move and draw are only evaluated at runtime. Of course, compiled Ada does not allow such constructions.

Notice the *semantics* of Ada remain intact: the most significant differences are the "line-at-a-time" level of interaction and the lack of an imposed "order."

Interactive Ada allows direct access to definitions as the following session illustrates. Origin, p1, p2 are declared to be points. One modifies the values of these variables by entering statements.

The important thing to notice is that the designer/programmer is free to enter statements, define variables, examine the values of variables, etc., without writing an entire program. The overhead of writing a main procedure, entering all routines in libraries, and so on, is eliminated during the experimental phase of prototype creation.

```
Surveyor> origin : point := (0,0,0);
Surveyor> p1 : point := (1,1,1);
Surveyor> p2 : point;
```

The following for loop negates each coordinate in p1, doubles it, and assigns it to p2.

```
Surveyor> for c in x..z loop
+                p2(c) := -p1(c) * 2;
+         end loop;
```

The following calls to move and draw invoke the specified procedure which is immediately executed.

```
Surveyor> move(origin);
Surveyor> draw(p1);
Surveyor> draw(p2);
```

The user now redefines draw to display the name of the point being drawn as a text string.

```
Survyeor> procedure draw(p : point)
+ is
+ begin
+     code("LG");
+     put_point(p);
+     put("Drawing to point (");
+     for c in x..z loop
+       put(p(c)); put(' ');
+     end loop;
+     put_line(")");
+ end;
```

The important features shown above are as follows:

1. The unit of interaction with Surveyor is an individual statement or declaration.

2. It is possible to delay the definition of a type.

3. It is possible to redefine a subroutine without destroying the current values of all global variables.

Table 2.1 on page 38 is a "directory" listing of the contents of a larger program. The program provides interfaces to a Tektronix 4129 terminal for creating and displaying 3D images [53]. It contains 24 variables, 6 types and subtypes, and 64 procedures and functions.

The type definitions, variable definitions, procedures, and functions are not stored in any particular order. (They are alphabetized for convenience only).

There is no apparent structure to the program. Each of the 64 procedures and functions is accessible directly as an interactive Ada command. The global variables are accessible as well.

2.8 Summary

This chapter surveys recent research on programming environments, and analyzes each in terms of its suitability for creating persistent prototypes. Existing programming environments fail to provide the proper blend of features necessary for prototyping *and* production use.

Surveyor supports interactive modification of Ada definitions in a line-at-a-time mode suitable for prototyping. Surveyor also supports strong typing which results in a more reliable prototype. Once exported, a prototype is reliable enough and structured enough so that further development in a production environment is not only possible, but desirable.

ambient : tekreal;	back_distance : integer;
color_smoothing : integer;	debug : boolean;
depoint_resolution : integer;	diffuse : tekreal;
eye_position : point;	front_distance : integer;
lightamt : tekreal;	lightplace : point;
obj_surf_disp : integer;	pan_dim : integer;
projection : integer;	rotate_radius : integer;
tekbase : integer;	uv_height : integer;
uv_width : integer;	view_norm : norm;
view_ref : point;	view_up : norm;
viewport_bounds : winbounds;	wantsolid : boolean;
window_bounds : winbounds;	tekload: boolean;
subtype norm;	subtype point;
subtype tekreal;	subtype winbounds;
type floatarr;	type intarr;
procedure ansi;	procedure box;
function chr;	procedure closeseg;
procedure cmap;	procedure cmap_entry;
procedure cmap_interp;	procedure cmaprange;
procedure code;	procedure coord_mode;
procedure dachars;	procedure daindex;
procedure dalines;	procedure davis;
procedure delseg;	procedure demess;
procedure depoint;	procedure draw;
procedure esc;	procedure fbox;
procedure fixsolids;	procedure flagging;
procedure ignore_deletes;	procedure killseg;
procedure lightsources;	procedure move;
procedure normout;	procedure openseg;
procedure outreal;	procedure overwindow;
function padd;	procedure page;
procedure panel;	procedure po;
procedure promptfile;	function psub;
procedure quadout;	procedure readsegs;
procedure renew;	function scalar_multiply;
procedure segment_transform_matrix;	procedure set_color_map;
procedure set_line_index;	procedure setlight;
procedure shapes;	procedure showmap;
procedure showmodel;	procedure solids;
function str;	procedure surfcol;
procedure tekint;	procedure tekmode;
procedure termcom;	procedure testbox;
procedure testpan;	procedure transdir;
procedure viewattr;	procedure viewport;
procedure viewtransform;	procedure window;
procedure winout;	procedure wirebox;
procedure wireframe;	procedure writesegs;

Table 2.1: Directory of Tektronix 4129 Package

Chapter 3

Collecting Dependency Information

> Many persons who are not conversant with mathematical studies imagine that because the business of [Babbage's Analytical Engine] is to give its results in numerical notation, the nature of its processes must consequently be arithmetical and numerical, rather than algebraical and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraical notation, were provisions made accordingly.
>
> — Ada Augusta, Countess of Lovelace (1844)
> as quoted by Donald E. Knuth, *Fundamental Algorithms* (1968)

As previously indicated, as definitions are changed, the initial ordering of those definitions in the interactive Ada environment is lost. Exporting a pro-

totype from Surveyor requires organizing these definitions into a more standard form. The export process can be broken down into two parts.

1. Collecting and organizing the dependency information already present in the environment, and

2. Performing a topological sort (including removal of cyclic references) to order the definitions.

In this chapter and Chapter 4, the details of the ordering process are presented. This chapter explains how the dependency information is collected. Chapter 4 explains the actual sorting algorithms.

The programmer can ask Surveyor to reorder his definitions. There are two reasons for reordering, as follows:

1. To organize the definitions in order to examine their structure.

2. To export the definitions to an external compiler.

3.1 Object Dependencies

A simple database called the *workspace* stores each definition as is entered. The workspace is divided into Ada packages. There are several predefined packages,

e.g., **standard** and **text_io**. The package **usr** is reserved and contains the objects defined by the user during a **Surveyor** session. Please refer to figure 3.1.

Each package contains a number of objects, either predefined or defined during a session. Objects often refer to other objects. For example, an object corresponding to an array declaration might refer to the predefined object **character** in package **standard**, as in

```
type carray is array(1..10) of character;
ca : carray;
```

Notice the implicit reference to the predefined object **integer** in the integer range 1..10. Also, the object **ca** depends on the object **carray**.

Over time, more objects are added to the workspace that depend on each other. Objects are *redefined* by entering a new declaration that replaces an existing one. For instance, changing **ca** from a character array to an integer will remove the dependency link from **ca** to **carray** and replace it with one to **integer**.[1] The new dependency graph is shown in figure 3.2.

---

[1]Overload resolution is not supported in **Surveyor**, except for some special case code to handle Ada's **put** and **get** I/O routines. Overloading identifiers does not introduce any special problems beyond the extra code for computing the entire signature (e.g., the name and argument types) of an identifier, and extra code for comparing two signatures. A declaration is redefined when a new declaration matches the signature of an existing definition.

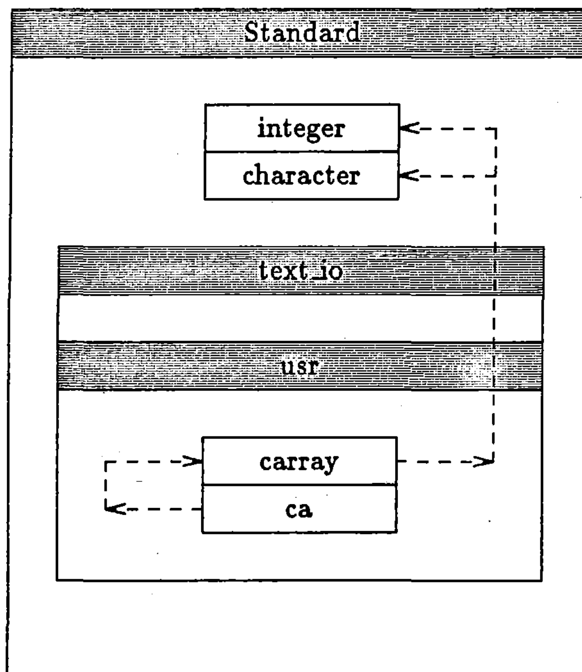


Figure 3.1: Layout of Surveyor Workspace

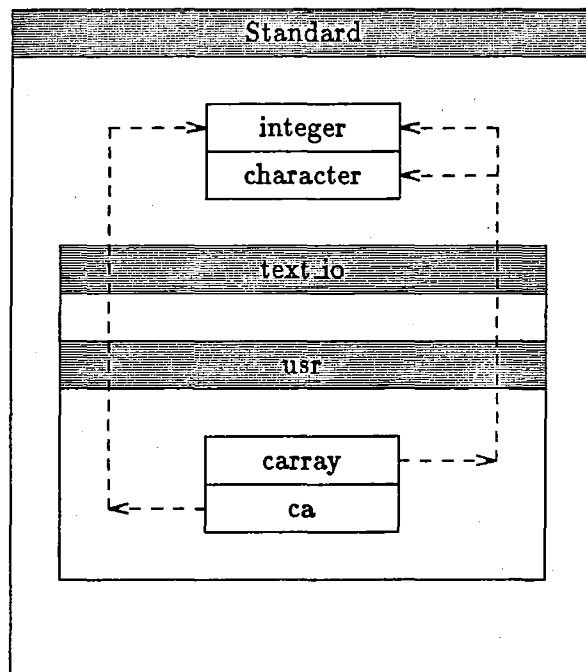


Figure 3.2: Workspace After Redefining **ca**

3.2 Symbols

A symbol consists of an object in the workspace and associated information, such as its name, its type, and other internal information. Symbols are manipulated by interactive Ada programs as well as by the interpreter. Symbols point to appropriate parts of the internal form; for instance, the symbol for a procedure points to the beginning of the internal form of the procedure.

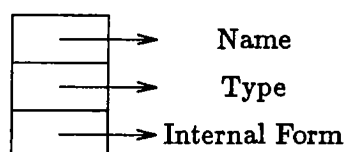


Figure 3.3: A Symbol

A predefined interactive Ada attribute, 'sym, provides access to symbols. This attribute, when applied to any identifier in Surveyor, returns the corresponding symbol (see figure 3.3). Symbols are predefined types in Surveyor. For example, fact'sym returns the symbol corresponding to the object fact. Some tools in Surveyor take symbols as arguments, e.g., the prettyprinter. The prettyprinter uses the symbol it is passed to find the internal form associated with the Ada object, and prettyprints it. For example,

```
> fact_symbol : symbol := fact'sym;
> pp(fact_symbol);
```

Symbol tables store collections of symbols within a particular Ada scope. The symbol table is indexed by the name of each symbol. The symbols are not stored in any particular order. Each can be accessed randomly by applying the attribute 'sym to its identifier. Other mechanisms exist for obtaining a list of the symbols present in any symbol table (see section 3.5).

Symbol tables contain additional information besides the symbols, as follows:

- a reference to the symbol that *owns*[2] the table,
- a static link to the enclosing table (scope), and
- a list of packages use'd in the scope.

Packages are represented by two tables. One contains declarations that are publicly available and the other contains private (local) definitions. For scoping purposes, the private table is considered lexically enclosed in the public table.

Ada allows packages to be use'd. The definitions in the package specification become visible at the point that the package is use'd. The use's list records the names of these packages. The use's list is not used during the collection process.

[2]For example, carray is in package usr; usr *owns* the table containing carray. The static link for *usr*'s table points to the table of the enclosing scope (the package standard in this case).

With clauses are unnecessary in Surveyor, because there is no concept of a library: a package is included as part of the workspace by reading in the text of the package with the built-in routine include.

3.3 Type Descriptors

Type descriptors (in the internal form) represent Ada types. These type descriptors form a graph of dependencies between all type definitions. The dependencies form a graph (rather than a tree) because Ada types sometimes refer to themselves via recursive record and access types. For example, consider a linked list:

```
type list;
type listp is access list;
type list is record
   value : integer;
   next : listp;
end record;
```

The type list depends on the type listp which depends on the type list.

Type descriptors are tree-like structures formed from *type components*. The top component in a type descriptor is called the *main component*; it specifies the general category of the type (e.g., array or record). All other components are called *subcomponents* and they specify various aspects of the general category. The types of main and subcomponents currently implemented in Surveyor are

listed in table 3.1. Notice that some top components are used as subcomponents when they occur as part of a larger type definition. For example, in:

```
type arr is array(1..10) of integer;
type arrpointer is access arr;
```

The type arr is both a top component and a subcomponent of arrpointer.

component	subcomponent(s)
named	symbol of base type
subrange	base type lower bound upper bound
enumeration	list of items lower bound upper bound
constrained array	base type list of index types
index	base type of index lower bound upper bound
derived	base type
access	base type
record	(optional) variant fields required fields

Table 3.1: Type Descriptor Components in Surveyor

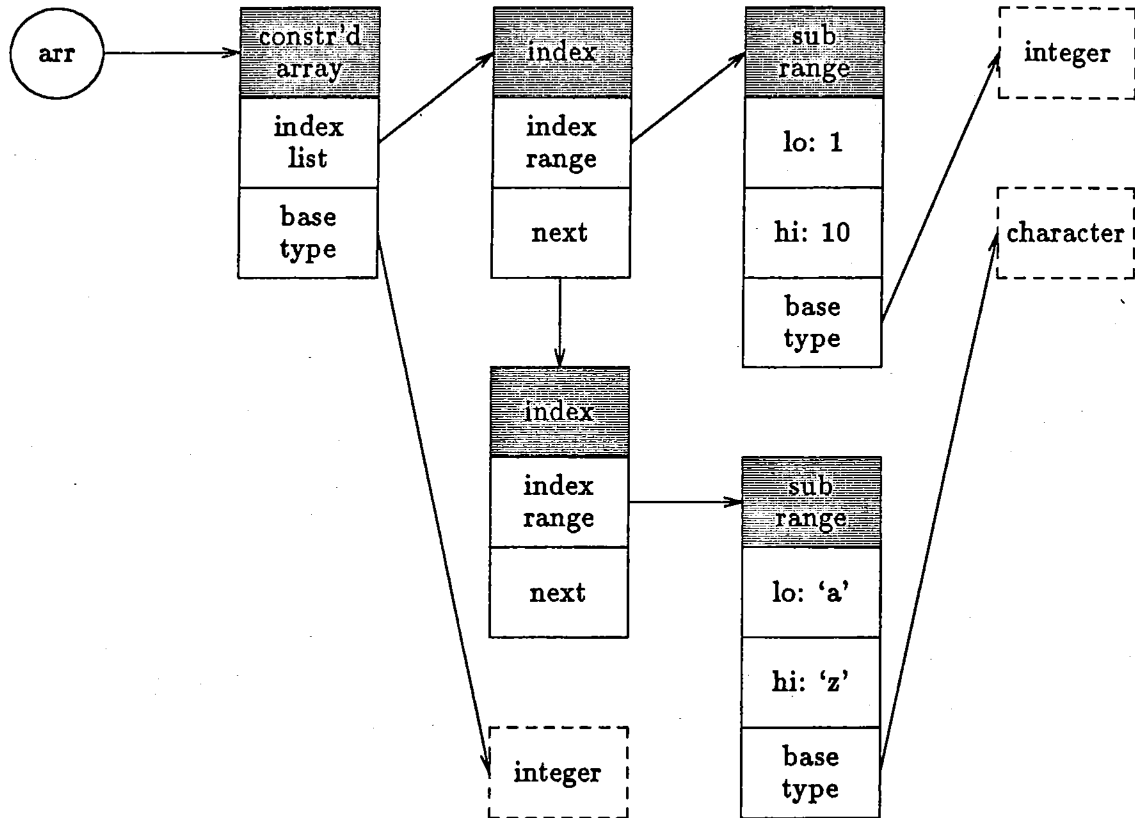
Figure 3.4 shows a picture of a sample type descriptor. The main component of the type descriptor is a constrained array. The subcomponents of the type descriptor are the list of index ranges and the base type. The index components have subcomponents consisting of the subranges for each index and the

base type of the index.

It is necessary, when tracing down the branches of a type descriptor, to eventually reach a *symbol* rather than another type descriptor component. The symbol makes the name of the base type explicit. It also makes redefining the meaning of a type much simpler because all references to the type are made indirectly through the symbol for the type. One can continue to analyze the subcomponents of *that* symbol's type, if necessary.

3.4 Internal Form

There are two "forms" of the internal form in Surveyor. The first is the *Bare Internal Form* (BIF). The BIF is a tree structure that resembles parse trees. The second is the Annotated Internal Form (AIF). The AIF is an annotated version of the BIF. The AIF is a graph structure and includes not only the original BIF trees, but also includes symbol tables and type descriptors. Generally speaking the AIF graphs are formed *around* the BIF trees, i.e., most explicit references point from the symbol tables to the internal form and not vice versa. The BIF trees are sometimes augmented further with back pointers to the symbol tables for efficiency of symbol table lookup operations, but this is an efficiency concern



type arr is array(1..10, 'a'..'z') of integer;
Note: The dashed boxes represent symbols.

Figure 3.4: Sample Type Descriptor

only, and is entirely optional.

The BIF contains sufficient information to derive a semantic description (the AIF) of an Ada program. This is reasonable, since the BIF is simply the parse tree for the original text with extraneous non-terminals removed.

The programmer interacts with Surveyor by entering fragments of Ada programs. The user interface delivers the text entered by the programmer to a parser. The parser in turn produces stripped-down syntax trees. These trees are adequate for recovering the original program text, convenient for interpretation, and a compact representation.

The Bare Internal Form is similar to *Diana* [10,11]. The Surveyor internal form has evolved to support interpretation of Ada programs instead of compilation.

The BIF is represented as a tree of variant records, varying according to each kind of node. Each variant record contains fields appropriate for that particular node. For example, an *elsif* node contains fields for the condition, the list of statements, and a pointer to the next *elsif* node in the chain.

Note: it is convenient to describe the internal form in terms of its concrete representation (as a tree of variant records) but this is done for convenience only. In fact, the concrete representation of the tree

structure of the code is unimportant since the trees are accessed via an abstract interface.

An Ada function to compute integer factorial, fact, is shown in figure 3.5. The text of fact's BIF tree is shown in figure 3.6.

```
function fact(arg : integer) return integer is
  begin
    if arg < 2 then return 1;
    else return arg * fact(arg-1);
    end if;
end fact;
```

Figure 3.5: Factorial Function

The text in figure 3.6 is structured in the following way:

- Quantities enclosed in square brackets represent nodes. A node may contain pointers to other nodes (or lists of nodes) or an atomic unit such as an integer or an identifier. The first element in a bracketed list is the node_op of the unit: *function*, *if*, *ident*, *return*, and *else* are node_ops.
- Lists are enclosed in braces. The elements in a list of structures do not necessarily have identical node_ops, however, elements are conceptually similar, as in a statement list or a declaration list or as otherwise suggested by

```
[function
    id:"fact"
    {[param {[id_decl id:"arg"]} [ident id:"integer"]]}
    [ident id:"integer"]
    {[if [< [ident  id:"arg" ] [int 2]]
            {[return [int 1]]}
     {[else {[return
                [* [ident id:"arg"]
                   [call [ident id:"fact"]
                         {[arg [- [ident id:"arg"] [int 1]]]}
                   ]
                ]
            ]}
     ]}
    ]}
]
```

Figure 3.6: BIF Tree for Factorial Function

the Ada syntax.

A BIF tree is augmented with semantic information to produce an *Annotated Internal Form* (AIF) graph. This semantic information includes scoping information and type information. The AIF graphs are generated by interpreting the declaration parts of Ada program fragments. The additional information makes it possible to convert the unordered fragments of interactive Ada into compiled Ada.

3.5 Mapping Functions

The AIF is a complex data structure incorporating many types of information. Scanning the different types of information is made easier by a set of *mapping functions*. These are analogous to the Lisp mapping functions, e.g., mapcar, but handle more than just a few cases (e.g., car, cdr, and atoms) as shown below.

There are three basic mapping functions in Surveyor.

1. Maptree is used to visit each node of a BIF tree (in order) and apply some specified function. Various uses of maptree include checking the size of a tree and copying all or part of a subtree.

2. Maptab is used to access all the symbols within a particular scope without

knowing their names. The export tool uses maptab to collect the symbols into a list.

3. Maptype is used to traverse a type descriptor tree. Maptype is used by the export tool to collect a list of the symbols used (or referenced indirectly) in a particular type definition. Type descriptors are the only structures that record certain dependencies. For instance, the subrange 1..10 implicitly references the type standard.integer, and this fact is recorded in the type descriptor (and therefore as part of the annotated internal form), but not in the bare internal form.

The mapping functions are large case statements recursively applied, where each case depends on a variant of the appropriate record. Each case is programmed to apply the supplied function to its node and (recursively) to all of its children.

3.6 The Collection Process

The sorting algorithms presented in Chapter 4 take, as input, a table of symbols and their dependencies. It is necessary to scan the definitions stored in the programming environment and build this table. This section presents a description

of that process.

The first step in this process is to collect a list of all the symbols in some specified scope. A function is supplied to maptab which builds a list of the symbols in the symbol table.

The second step is to collect, for each symbol, a list of symbols referenced by the definition of that symbol. There are three classes of definitions, each a little different from the others. These are:

1. type definitions and object declarations;

2. constants, constraints, and initializers; and

3. procedures, functions, and packages.

The first case uses type descriptors. The type descriptor for a type definition explicitly includes references to other symbols used in the type definition. The process of collecting the symbol references is implemented via maptype analogous to the way the original list of symbols in the scope is collected. A function is applied to every component of the type descriptor. This function collects a list of symbols referenced in various components of the type descriptor.

The second case augments the first. Initializers and constraints are Ada expressions represented by BIF trees. The BIF trees reference variables, functions,

and constants. It is necessary to know which functions, constants, and variables are referenced, in order to ensure each is declared before it is used.

A procedure, function or package contains a local symbol table. The process described herein is applied recursively to the symbols in the local table. In addition, the BIF tree for the procedure, function, or package *body* is analyzed for references to other variables, constants, packages, procedures, and functions. The BIF trees for the appropriate procedure, function, and package bodies are traversed, and a function is applied at every node. This function adds the symbols referenced by various nodes to a list. In this way a list of depended-on symbols is generated.

3.7 Example

The following sample set of definitions implement a hash table. The hash table is implemented as an array of bins, where each bin contains a linked list of the numbers that hashed to that bin. The definitions are shown entered in the order they would be entered by a programmer, e.g., with some redefinition of previously entered definitions.

The programmer begins by declaring the size of the hash table.

```
Surveyor> size : integer := 16;
```

The programmer next declares the link listed structure.

```
Surveyor> type list;
Surveyor> type listp is access list;
Surveyor> type list is record
+   next : listp;
+   val : integer;
+ end record;
```

The hash table itself is an array of pointers to these linked lists.

```
Surveyor> type table is array(1..size) of listp;
```

At this point, the programmer changes the size of the variable used to define the hash table. This introduces a logical inconsistency into the code.

```
Surveyor> size := 32;
```

The programmer examines the values of the indices of **table**, which retain their old values of 1 and 16. They retain their value because while the redefinition of types is automatic, it is not necessarily reasonable to change existing instances of a type. In particular, the problem of what to do with the old value of a variable is a problem for future research (see Chapter 5).

```
Surveyor> put(table'first);
1
Surveyor> put(table'last);
16
Surveyor> put(size);
32
```

The programmer defines a variable mid as the midpoint of the table. Mid is defined in terms of the actual table indices, rather than the variable size.

```
Surveyor> mid : integer := table'last / 2;
Surveyor> put(mid);
8
```

The programmer now defines a variable half in terms of the size variable. Notice that half is given its value via an initializer. When half and size are later exported, their initial values are the value specified in the initializer, and *not* their current values. In this way the programmer is given explicit control over their initial state.

```
Surveyor> half : integer := size / 2;
Surveyor> half
16
```

Finally, two functions to enter and look up a number are entered by the programmer. Notice the function lookup automatically enters the number if it is not already in the table.

```
Surveyor> function lookup(val : integer) return boolean is
+         where : integer := val / table'last + 1;
+         l : listp := table(where);
+ begin
+         while l /= null loop
+                if l.val := val then
+                         return true;
+                 else l := l.next;
+                 end if;
```

```
+          end loop;
+          enter(val);
+          return false;
+ end;

Surveyor> procedure enter(val : integer) is
+          where : integer := val / table'last + 1;
+          l : listp := new list(table(where), val));
+ begin
+          table(where).next := l;
+ end;
```

The first step of the collection process is to collect all symbols in the scope into a list. Table 3.2 shows this list of symbols.

enter
half
list
listp
lookup
mid
size
table

Table 3.2: List of Symbols

Table 3.3 lists the information collected from the type descriptors of those symbols with a specific type. (The specific type of a function is the return type of the function.)

Table 3.4 lists the various dependencies collected from initializers and constraints. For example, the bounds of the array table are based on the simple

Symbol	Type	Depends On
half	simple	integer
list	record	listp, integer
listp	access	list
lookup	function	boolean
mid	simple	integer
size	simple	integer
table	array	integer, listp

Table 3.3: Dependencies from Type Descriptors

expression size.

Symbol	Type	Depends On
half	simple	size
mid	simple	table
size	simple	integer
table	array	size

Table 3.4: Initialization and Constraint Dependencies

The local definitions in the subroutines lookup and enter contain references to other objects in the enclosing scope. It is necessary to find every reference made in each routine. References in subroutines and packages are found in local definitions as well as in statements. Table 3.5 lists the dependency information obtained analyzing the statements in enter and lookup. Note: the definitions inside lookup and enter are already in order.

Table 3.6 summarizes all the dependency information collected above.

Symbol	Type	Depends On
enter	procedure	integer, table, listp, list
lookup	function	integer, boolean, table, listp, enter

Table 3.5: Dependencies in lookup and enter

Symbol	Type	Depends On
enter	procedure	integer, table, listp, list
half	simple	integer, size
list	record	listp, integer
listp	access	list
lookup	function	integer, boolean, table, listp, enter
mid	simple	integer, table
size	simple	integer
table	record	integer, size, listp

Table 3.6: Summary of Dependencies

Figure 3.7 shows a picture of the complete dependency graph. The arrows start at the center of a dependent symbol and point to the nearest corner of the depended-on symbol. Integer and boolean are defined in the enclosing scope.

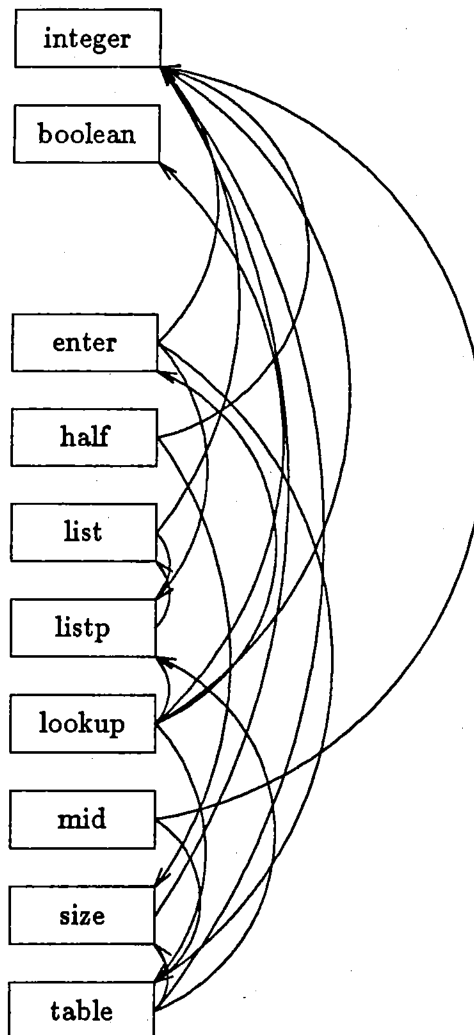


Figure 3.7: Dependency Graph

Chapter 4

The Export Tool

> The problem of topological sorting ... is to find a way to arrange [things] ... so that no term is used before it is defined. Analogous problems arise in writing programs to process the declarations in certain assembly and compiler languages...
>
> — Donald E. Knuth, *Fundamental Algorithms* (1968)

The *export tool* looks at the symbols present in a particular scope and the dependencies that exist between them and prints an ordered listing. The export tool is an important part of the Surveyor architecture; it is *extremely* difficult to capture the results of an interactive session, by hand, in a form usable in a compiler-based environment. The user would have to prettyprint the definitions of all the symbols in the workspace, and laboriously compare each definition for references to other definitions.

Topological sort is a well-known technique [29, pp. 258–268][1] for arranging a directed acyclic graph (DAG) into a linear order. Topological sort would be applicable directly to the problem of sorting a symbol table if the partial ordering between symbols was acyclic. Alas, symbols frequently have recursive definitions. There are two cases in Ada where such recursive definitions are allowed:

1. An access type, A, may refer to a record, R, which contains a field whose type is A; and,

2. A subprogram definition, F_1, may contain a reference to a subprogram, F_2, which in turn refers to F_1.

The four algorithms presented here implement a form of topological sort for Ada definitions, where the cyclic dependencies are removed by generating appropriate "forward" definitions, as follows:

1. The target of an access type is declared before the access type as an incomplete type definition, and

[1]Knuth presents a specific implementation in addition to a high-level description. Knuth's implementation is tailored to sorting nodes which are labeled by small integers. Bentley [6] provides an alternative implementation based on associative arrays in AWK [3]. Bentley's implementation works for nodes labeled with arbitrary strings. The description here is drawn from both [29] and [6], but ignores the details of the concrete representation.

2. Subprogram stubs are generated for recursively referenced subprograms.

Algorithm TGEN constructs a table used by the other algorithms for finding the next symbol to print. Algorithm TSORT is the standard topological sort; the only modification is to invoke algorithm BREAKCYCLE when a cycle is detected. Algorithm BREAKCYCLE is charged with breaking one or more cycles so that algorithm TSORT can continue.

The algorithms themselves are presented in pseudo-code. Various list manipulating routines are used:

- append(L, I) — returns a new list consisting of the old list L with item I catenated onto the end.
- tail(L) — returns the tail of the list L, i.e., everything except the first item of the list.
- head(L) — returns the first element of the list L.
- remove(L, I) — returns a new list with all occurrences of the item I removed from L.

Comments are introduced with two hyphens ("--") and are terminated by the end of the line. The word "continue" signifies that control returns to the top of the loop in which it is embedded.

4.1 Algorithm TGEN

Purpose

Algorithm 1 generates a table containing a count of the predecessors of each symbol and the set of its successors. This table, called the table of predecessor counts and successor sets (PCSS), improves the running time of the topological sort algorithm. Without this table, the algorithm must search the entire graph each time for a node with no predecessors. With the table, the running time for the basic topological sort is proportional to the number symbol pairs, which is within a constant factor of optimal [6, page 574].

One additional transformation is included. Any direct cycles in the graph are eliminated while the table is built. The offending dependency is simply ignored. This situation only arises when directly recursive subroutines exist in the declarations being sorted, and provides no useful information.[2]

Inputs, Outputs, and Local Data

The input to algorithm TGEN is a list of symbol pairs, *(symbol, depended on symbol)*. These symbols pairs are gathered using the techniques in Chapter 3.

[2]Directly recursive subroutines do not require generating any "forward" references.

Only one symbol *table* is sorted.

One of the outputs of algorithm TGEN is a list of external scopes (symbol tables) referenced in the scope being sorted. The *parent symbol table* for a symbol is defined to be the symbol table containing it. While building the table of predecessor counts and successor sets (PCSS), the parent of each symbol is examined. If the symbol's parent table is different from the table presently being sorted then the reference is *out of scope*, i.e., to some external symbol table. The name of the owner of the table is saved and can later be made part of an Ada *with* clause.

Below, S and P refer to the elements in the symbol pair (symbol, depended on symbol) mentioned above. For each, the *.predct* field will contain the count of immediately preceding symbols, and the *.succlist* field will contain a pointer to a list of successor symbols. An additional field, *.marker*, is used later by algorithm TESTCYCLE, and is initialized by TGEN to zero.

Algorithm

```
for each (S, P) pair
    S.marker := 0
    P.marker := 0
    if S = P then
        continue
    end if
```

```
    if is_local(S)³ then
        if is_local(P) then
            S.predct := S.predct + 1
            S := append(P.succlist, S)
        else
            with_list := append(with_list, P)
    else
        with_list := append(with_list, S)
    end if
end for
```

Comments on Algorithm TGEN

The table of predecessor counts and successor sets is generated as described in [6], with the following exceptions:

1. The exact data structures are not specified; the reader is refered to [29, page 261] or [6, page 574].

2. The generation of the with_list is a new addition, made necessary because the sort is being done in the context of an Ada program.

---

[3] The routine *is_local*, not defined here, simply checks if the parent of the symbol is the symbol table presently being sorted, and returns *true* if it is, otherwise *false*.

4.2 Algorithm TSORT

Purpose

Algorithm TSORT does the main processing, printing declarations until a cycle makes printing further symbols impossible.

Inputs, Outputs, and Local Data

The PCSS table generated by algorithm TGEN is the input for TSORT. It scans for all symbols S_n with a predecessor count of zero; each symbol found can be output at once. All successors to S, called T_n, have their predecessor counts decremented. All of the T_ns whose predecessor counts become zero are the next candidates for output. A queue, Q, is used to remember the order in which the counts become zero. If, after the main processing is done, any symbols remain which have not been output, there is a cycle in the graph.

The output of this routine is a printed list of symbols in the scope, with the partial ordering intact, i.e., each symbol is defined before it is used.

Below, S and T are references to symbols defined in the PCSS table built by TGEN. The fields *.predct* and *.succlist* have the same meaning as in TGEN. The variable N is used to count how many symbols have been successfully sorted.

N is initially equal to the number of symbols in the table, and decremented each time one is printed. At least one cycle is present when Q is empty and N is non-zero.

Algorithm

```
Q := NULL
-- First find all symbols with no predecessors
-- and add them to Q.
for each symbol S
    if S.predct = 0 then
        Q := append(Q, S)
    end if
end for
-- Process contents of Q
while N /= 0
    while Q /= NULL
        T := head(Q)
        Q := tail(Q)
        N := N - 1
        output T
        for each S in T.succlist
            S.predct := S.predct - 1
            if S.predct = 0 then
                Q := append(Q, S)
            end if
        end for
    end while
-- One or more cycles are present if N is non-zero.
    if then N /= 0 then
        call algorithm BREAKCYCLE
    end if
end while
```

Comments on Algorithm TSORT

This algorithm also is directly from [6], with the same stipulations as for algorithm TGEN. The only addition is that instead halting with an error message when a cycle is detected, algorithm BREAKCYCLE is invoked to resolve any circular dependencies. When algorithm BREAKCYCLE returns to algorithm TSORT, the queue will be non-empty, and the algorithm can continue to find and output symbols with zero predecessor counts until it is blocked by another cycle or it completes.

Algorithms TGEN and TSORT are normally combined into a single algorithm; they are separated here to emphasize the impact of language constraints on the sorting process.

4.3 Algorithm BREAKCYCLE

Purpose

The purpose of algorithm BREAKCYCLE is to add at least one new symbol to the queue Q so that algorithm TSORT can continue to unwind the dependency graph. Cycles arise in the dependency graph for two reasons.

1. A record type may contain a field with an access type which points, directly or indirectly, to the same record.

2. Subroutines and functions may have recursive references to other subprograms.

The dependency relationships between subroutines can be arbitrarily complex. A function or procedure may call any other function or procedure. Consider, for example, a function F_1 that calls F_2, F_3, F_4 and F_5, each of which recursively calls all of the others. Direct cycles are avoided (in algorithm TGEN) by never adding a symbol to its own successor set.

It is necessary to break one or more cycles by printing one or more forward declarations. Cycles are found by exhaustively analyzing the remaining symbols in the dependency graph. When a cycle is found, it is broken with a forward declaration for some symbol S, and the dependency counts of all symbols in *S.succlist* are reduced. If any of these counts go to zero, algorithm TSORT can continue. Otherwise, another cycle must be found and broken.[4]

---

[4]The entire set of algorithms can be replaced by a simplistic algorithm which prints all possible forward declarations (an incomplete type definition for the target of every access type, and a stub for every subprogram), and then prints the symbols in some random order. The resulting listing isn't very interesting, although this approach may have some value if an *alphabetical* listing is desired.

Inputs, Outputs, and Local Data

The input to algorithm BREAKCYCLE is the PCSS table of TSORT, where all symbols with zero *.predct*'s have been output, and some symbols with non-zero *.predct*'s remain. This situation only arises when there is a cycle in the dependency graph [6, page 574].

Two side-effects of algorithm BREAKCYCLE is a non-empty queue Q and a modified PCSS table. Additionally, one or more forward declarations are output.

BREAKCYCLE has a local integer variable, M, initially 0, which is incremented and then passed to TESTCYCLE. M is used by TESTCYCLE to uniquely mark each symbol it visits (see the description of TESTCYCLE in the next section).

Algorithm

```
-- Continue breaking cycles until a symbol is added to Q
M := 0
for each symbol S
    if Q /= NULL
        -- Terminate when a symbol has been added to Q
        return
    else
        -- Skip S if it has already been output
        if S.predct = 0 then
            continue
        end if
        M := M + 1
```

```
        if SUITABLE(S)[5] and then TESTCYCLE(S, S, M)[6] then
            output "forward" declaration for S
            for each T in S.succlist
                T.predct := T.predct - 1
                if T.predct = 0 then
                    Q := append(Q, T)
                end if
            end for
        end if
    end if
end for
-- Something is seriously wrong if control gets to here -
-- all symbols were looked at without breaking a cycle.
```

Comments on algorithm BREAKCYCLE

This particular algorithm for finding cycles (testing each symbol individually) was chosen for understandability rather than computational efficiency. The running time is improved by invoking it only when necessary, i.e., algorithm TSORT cannot continue, and so only the minimum number of symbols are checked for their presence in a cycle.

The reader is refered to [43] for an efficient and general directed graph or-

---

[5]SUITABLE is a routine, not defined here, which merely checks if S is a procedure, function, or record definition. As explained in the comments for BREAKCYCLE, it is only necessary to test if these kinds of symbols are involved in a cycle, since these are the only kinds of symbols which can be declared forward.

[6]TESTCYCLE checks if S is involved in a cycle. It is defined in the next section.

dering algorithm (without any programming environment specific details). Also, it is desirable in this setting to find only a single cycle, rather than all cycles. In this way, algorithm TSORT will continue to run with only a small number of forward declarations being produced.

Only two types of symbols have corresponding forward declarations, namely, records and subprograms. There is little point in analyzing a symbol which is not of one of these types, hence the suitability test above.

Various optimizations are possible, most notably remembering other cycles as they are encountered[49, page 44].

4.4 Algorithm TESTCYCLE

Purpose

Algorithm TESTCYCLE returns true if its first argument S, is a part of a cycle. The test for circularity is made by traversing the paths away from S and checking if any of them returns to S. TESTCYCLE is implemented by recursively analyzing the nodes in the successor set for S. TESTCYCLE avoids interior cycles (i.e., cycles potentially existing within a particular path) by marking the symbols it has visited with a unique identifier (an integer).

Inputs, Outputs, and Local Data

The input to TESTCYCLE is the symbol to be tested, S; the current symbol along a particular path, S' (initially $S' := S$); and a unique marker, M. TESTCYCLE uses the successor set of S' as the set of paths to follow when searching for a cycle containing S. Each symbol T in the successor set of S' contains a *.marker* field. The value of this field is initialized to zero by TGEN. TESTCYCLE checks each *T.marker*. If *T.marker* $= M$ then TESTCYCLE has already visited that particular node; otherwise, TESTCYCLE sets *T.marker* $:= M$ and continues. The calling routine (algorithm BCYCLE) ensures that M is unique for every top-level (i.e., non-recursive) invocation of TESTCYCLE by passing in steadily increasing integers.

The only output from TESTCYCLE is the return value of **true** or **false**.

Algorithm

```
For each symbol T in S'.succlist
    if T = S then
        return true
    elsif T.marker = M then
        return false
    else
        T.marker := M
        if TESTCYCLE(S, T, M) then
            return true
        end if
```

```
    end if
    return false
end for
```

Comments on Algorithm TESTCYCLE

A symbol S is contained in a cycle only if there is a path along some set of arcs emanating from S, through one or more intermediate nodes $T_1 \ldots T_n$, which terminates at S. (There are never any cycles with no intermediate nodes, as these are removed when the PCSS table is built by TGEN.) In the worst case, TESTCYCLE exhaustively traces every path originating at S, so if a cycle exists, TESTCYCLE must find it. The use of the *.marker* field in each symbol ensures that TESTCYCLE will never get caught in an endless loop. Notice that if S has no successors, TESTCYCLE returns false immediately, as the *for* loop will never execute.

4.5 Examples

This section presents two examples.

The first example continues with the dependency data gathered in Chapter 3. For reference, table 3.6 is reprinted below as table 4.1.

The first task is to compute the table of predecessor counts and successor sets for those symbols. Table 4.2 shows the results of this processing.

Working through the example, algorithm TSORT detects a cycle after printing size and half. Algorithm BREAKCYCLE is activated. Algorithm BREAKCYCLE removes the cyclic dependency between list and listp. An incomplete type definition for list is printed. Processing returns to TSORT. Listp is printed and the predecessor count of list is reduced to zero. In like fashion the rest of the symbols are printed.

```
size : integer := 16;
half : integer := size / 2;
type list;
type listp is access list;
type list is record
        next : listp;
        val : integer;
    end record;
type table is array(1..size) of listp;
mid : integer := table'last / 2;
procedure enter ...
function lookup ...
```

Figure 4.1: Ordered Interactive Ada Fragments

Figure 4.1 shows the text of the ordering for these sample definitions. The second assignment to size is ignored: the definition with an initial value is as-

Symbol	Type	Depends On
enter	procedure	integer, table, listp, list
half	simple	integer, size
list	record	listp, integer
listp	access	list
lookup	function	integer, boolean, table, listp, enter
mid	simple	integer, table
size	simple	integer
table	record	integer, size, listp

Table 4.1: Summary of Dependencies

Symbol	Type	Predecessor Count	Successor Set
enter	procedure	2	lookup
half	simple	1	
list	record	1	*(listp)*
listp	access	1 *(0)*	enter, table, list, lookup
lookup	function	3	
mid	simple	1	
size	simple	0	table, half
table	array	2	enter, mid, lookup

Table 4.2: Predecessor Counts and Successor Sets

sumed to be the correct default value for size. The inconsistency between the variables mid and half is removed: they now initialize to the same value.

Notice the definition of half was moved forward. While not necessary, moving it does improve the readability of the definitions. In general, many orderings are possible for a complicated graph. The dependencies between objects specify a *partial ordering* only.

Figure 4.2 shows the dependency graph in in sorted order.

The second example focuses on unwinding a complex sequence of cycles. A picture of the graph is show in figure 4.3. The arrows point to successor nodes.

Node 1 will always be output first, since it has no predecessors. The predecessor count of node 2 is reduced from four to three. At this point, there are no more nodes to output. A node is chosen at random, say node 4. Node 4 is in a cycle, and so it is declared forward. The predecessor counts of nodes 3, 2 and 5 are decremented. Node 5 is output, as it depended only on node 4. Node 6 is output, since it depended only on node 5. Again, there are no more candidates, so another cycle is broken. Suppose node 3 is chosen. It is part of a cycle, so it is declared forward. The predecessor counts of nodes 2 and 4 are decremented. Node 2 is output, followed by node 3, and finally node 4. This information is summarized in table 4.3.

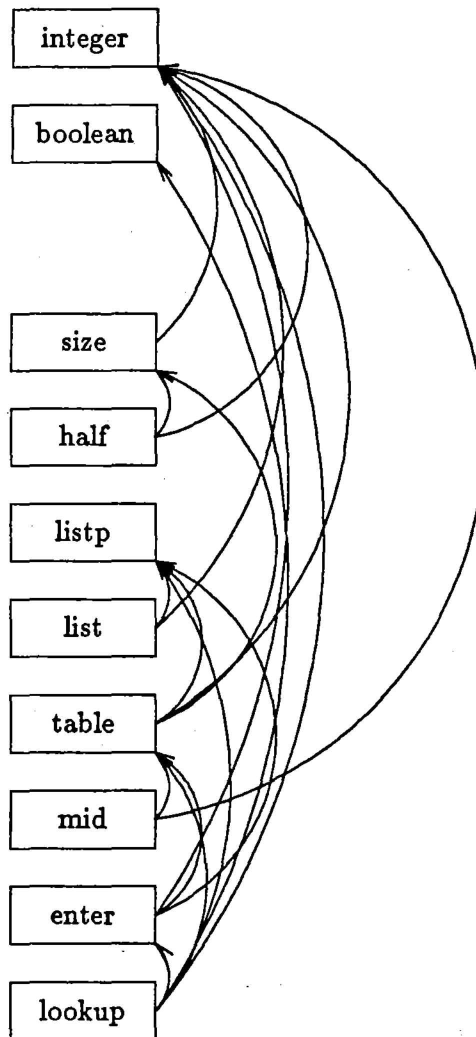


Figure 4.2: Sorted Dependency Graph

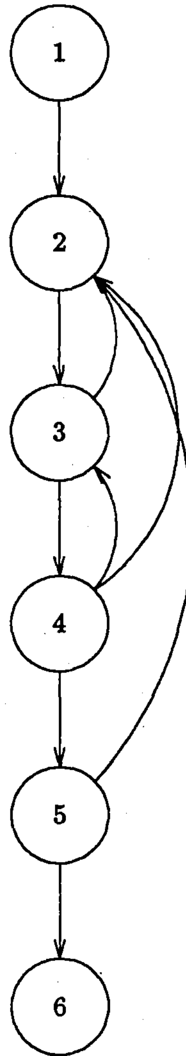


Figure 4.3: Sample Cyclic Graph

The exact ordering depends on the order nodes are chosen to break cycles. An interesting problem which remains to be solved is to determine which nodes are the best candidates to analyze first, thus minimizing the number of forward references. Table 4.3 also shows the ordering resulting from different choices of nodes.

Node Order	Sort Results
4, 5, 6, 3, 2, 1	1, 4_f, 5, 6, 3_f, 2, 3, 4
5, 4, 3, 2, 1, 6	1, 5_f, 6, 4_f, 5, 3_f, 4, 2, 3
1, 2, 3, 4, 5, 6	1, 2_f, 3_f, 4, 3, 5, 2, 6

Note: The notation n_f indicates a forward declaration.

Table 4.3: Results of BREAKCYCLE

Chapter 5

Recapitulation

> The obvious is that which is never seen until someone expresses it simply.
>
> — Christian Morganstern

This chapter indicates some lessons learned and suggests some areas for future research.

5.1 Lessons Learned

Linear Ordering

Ada specifies an *order of elaboration* for the definitions within a particular scope. This order of elaboration is the single greatest impediment to using Ada as a

prototyping language. The export tool presented here solves this problem, but the question remains, why is it necessary to have the definitions follow such a strict order of elaboration?

There are three answers to this question.

1. It is easier to write a compiler that processes each declaration, including any initializers, as it is seen.

2. The programmer can write declarations in the same way he/she writes statements, with constraints and initializers depending on previously defined statements.

3. The scoping rules allow access to global definitions if they are used before the local definition that hides it.

There are several objections to each of these reasons for imposing an order of elaboration on the definitions within a scope.

Regarding the first item, production quality compilers frequently read entire program blocks before generating any code. This allows optimization of the layout of stack frames as well as much greater freedom for code movement and optimization. Compilers that use the *Diana* internal form generally create the internal representation before any processing starts.

Regarding the second item, when prototyping, it is frequently useful to use a definition before it is defined textually in the source code. This capability eliminates the need for forward declarations and incomplete type definitions. This "unordered" approach is clearly beneficial in a prototyping environment, and may be useful in a production environment as well. It is not clear that because a programming language includes strong typing that it needs a strict order of elaboration as well.

Finally, global variables are always accessible in Ada by qualifying the name of the global variable directly (e.g., `proc1.var2`). Altering the scoping rules so that an identifier is available for use preceding its definition actually reduces the ambiguity: it is always necessary to qualify the global definition and therefore document the fact it is hidden by a local definition with the same name.

The advantage of removing this restriction is that one aspect of the problem of getting everything "in the right order" is removed. The statements in a program obviously must be in the right order. There is no compelling reason why the declarations within a scope must follow a linear ordering. As this dissertation indicates, the complexity of keeping the declarations in the right order is significant and cannot be ignored.

Experimental Programming Environments

One important lesson learned is that it is enormously beneficial to have access to the source code for an existing programming environment. Some environment research necessitates creating an entire new programming environment from scratch. In such unfortunate situations, the amount of work involved merely to get up to speed is incredible. Whenever possible, it is advisable to adapt an existing environment. In this way, the new research is more tightly focused: it is not necessary to reinvent the wheel (actually, parsers, prettyprinters, interpreters, etc.). In addition, it is easier to differentiate new research results from previous work.

The Arcturus environment began as a simple Ada interpreter. Over time, all sorts of experimental features were added, and integrated into one cohesive whole. It became difficult after awhile to say exactly what Arcturus was and what it was not, and in particular, to say what each student's contribution had been. For his research on persistent prototyping, the author removed whole pieces of the environment toolset. The result was a return, of sorts, to a simple interpreter based system which became the platform for further, individualized work. The pace of programming environment research was greatly increased with the advent of the Interlisp system, where each user of the system had access to the source

code. Further Ada programming environment research will have much more value if a programming environment platform, like Arcturus or Surveyor, is widely available. Adding a new Lisp-specific tool to Interlisp was relatively simple, because the Lisp language was simple. Adding an Ada-specific tool to Arcturus is much more difficult: the semantics of the language are more complicated.[1] This difference in complexity serves to underscore the need for a solid research platform.

5.2 Future Work

Automatic Hierarchical Organization

Prototype programs are generally organized in one large scope. The intent is to keep as many definitions visible as possible so that modifications to the structure of the system are as unconstrained as possible. The graph of dependencies between fragments of the code will contain an implicit hierarchical organization. It is desirable to make this implicit hierarchy explicit when exporting to a production environment.

There are two key problems that need to be solved.

[1]As a hint of the complexity, consider that the syntax of Lisp is easily described in ten to fifteen BNF rules; Ada requires about 140 rules.

1. A means for grouping related definitions into packages needs to be discovered.

2. Any mechanism for breaking the dependency graph into chunks is likely to produce several alternatives. A method of displaying the choices so the user can select between them is needed.

Retaining the Value of a Redefined Type

It is sometimes desirable to retain the value of an object even when its type definition changes. This unusual condition occurs only in an interactive environment with strong typing. Presumably all instances of the type would lose their current values. This need not be the case. Sometimes the new type definition is "close" to the previous type definition. Examples are changing the length of an array, or adding a field to a record. There is no reason the old values of the objects shouldn't be retained if possible. The benefit is that the values of complex data structures do not have to be recreated by the programmer.

There are two main areas of research necessary:

1. A detailed classification of the varieties of types possible and the various changes they might go through, and

2. An exact definition of the "closeness" of two type definitions needs to be developed.

Another benefit of research in this area will be the development of a means of automatic type coercion between arbitrarily complex but "close" user defined data types.

5.3 Summary

This dissertation discusses the problem of creating *persistent prototypes*: prototypes created in a rapid prototyping environment and exported to traditional compiler based programming environments for further development and refinement. It shows that only languages with strong typing are suitable for building prototypes which evolve into a mature system.

The most significant difficulty with automatically organizing prototype interactive Ada programs is collecting the dependency data. One approach, based on the use of type descriptors as carriers of dependency information, is presented. In this approach, the basic structure of the program is viewed as a dependency graph between every object (type, constant, subtype, subprogram, etc.) in the prototype system. The unstructured prototype implementation is translated to

a structured version of the same program. There are two benefits from this translation:

1. The translated version documents the relationships between definitions in the prototype.

2. The resulting translation is compilable in standard compiler based programming environments. The scaffolding of the interactive development environment is completely stripped away.

The basic algorithm for linearizing the structure of the prototype is the topological sort by Knuth. The algorithm is modified to handle circular dependency relationships as they arise in Ada programs.

The approach here is *proactive.* The programmer specifies a minimum amount of structuring information (e.g., strong typing of objects) and the system imposes all other redundant constraints (e.g., a specific program ordering) automatically. This approach improves upon the present approach in which consistency constraints are *enforced* and the programmer is left to the task of satisfying those constraints by himself.

References

[1] Ada Joint Program Office, *Ada Programming Language Reference Manual*, ANSI/MIL-STD-1815A-1983, 1983.

[2] N.I. Adams and J.A. Rees, *The T Manual Pre-release Edition*, Computer Science Department, Yale University, July 1982.

[3] A.V. Aho, B.W. Kernighan and P.J. Weinberger, *AWK — a pattern scanning and processing language (second edition)*, Unix Programmer's Guide, Volume IIB, Bell Laboratories, Murray Hill, 1979.

[4] Association for Computing Machinery, *Collected Algorithms of the ACM*, Association for Computing Machinery, New York, 1978.

[5] D. Barstow, *Rapid prototyping, automatic programming, and experimental sciences*, ACM Software Engineering Notes, 7, 5 (December 1982), pp. 33–34.

[6] J. Bentley, *Associative arrays*, Communications of the ACM, 28, 6 (June 1985), pp. 570–576.

[7] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, 1981.

[8] B.W. Boehm, T.E. Gray and T. Seewaldt, *Prototyping vs. specifying: a multiproject experiment*, IEEE Transactions on Software Engineering, 10, 3 (May 1984), pp. 290–302.

[9] W.G. Brown, ed., *Reviews in Graph Theory*, American Mathematical Society, Providence, 1980.

[10] K.J. Butler III, *Draft Revised Diana Reference Manual*, Tartan Labs, Inc., Pittsburgh, June 1982.

[11] K.J. Butler III and A. Evans, *Interim Diana Report*, Tartan Labs, Inc., Pittsburgh, October 1982.

[12] T.E. Cheatham, J.A. Townley and G.H. Holloway, *A System for Program Refinement*, Technical Report 5-79, Research Computing Center, Harvard University, Cambridge, August 1979.

[13] R.B.K. Dewar, et.al., *The NYU Ada Translator and Interpreter*, Courant Institute of Mathematical Sciences, New York University, New York, 1980.

[14] N.M. Delisle, D.E. Menicosy and M.D. Schwartz, *Viewing a programming environment as a single tool*, ACM Software Engineering Notes, 9, 3 (May 1984), pp. 49–56.

[15] L.P. Deutsch and E.A. Taft, *Requirements for an experimental programming environment*, CSL-80-10, Xerox Palo Alto Research Center, Palo Alto, June 1980.

[16] V. Donzeau-Gouge, G. Kahn, B. Lang and B. Melese, *Documents structure and modularity in Mentor*, ACM Software Engineering Notes, 9, 3 (May 1984), pp. 141–148.

[17] C.N. Fisher, et.al., *The POE language-based editor project*, ACM Software Engineering Notes, 9, 3 (May 1984), pp. 21–29.

[18] P. Fritzson, *Preliminary experience from the DICE System, a distributed incremental compiling environment*, ACM Software Engineering Notes, 9, 3 (May 1984), pp. 113–123.

[19] M. Franz, *The state of C interpreters*, PC Tech Journal, 4, 5 (May 1986), pp. 153–163.

[20] J.D. Gannon, *An experimental evaluation of data type conventions*, Communications of the ACM, 20, 8 (August 1977), pp. 584–595.

[21] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, John Wiley & Sons, Inc., New York, 1982.

[22] J. Holloway, G.L. Steele Jr., G.J. Sussman and A. Bell, *The Scheme-79 Chip*, AI Memo No. 559 (also EE&CS Integrated Circuit Memo No. 80-6), Artificial Intelligence Laboratory, Massachusetts Institute of Technology, January 1980.

[23] K. Jenson and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, New York, 1974.

[24] S.C. Johnson, *Lint, a C program checker*, Unix Programmer's Guide, Volume IIB, Bell Laboratories, Murray Hill, 1979.

[25] S.C. Johnson, *Yacc: yet another compiler-compiler*, Unix Programmer's Guide, Volume IIB, Bell Laboratories, Murray Hill, 1979.

[26] B.W. Kernighan and J.R. Mashey, *The Unix programming environment*, IEEE Computer, 14, 4 (April 1981), pp. 12–24

[27] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978.

[28] R.O. Klefstad III, *Uniform User Interface for a Programming Environment*, Ph.D. dissertation in preparation, Computer Science Department, University of California, Irvine, 1986.

[29] D.E. Knuth, *The Art of Computer Programming vol 1: Fundamental Algorithms*, Addison-Wesley Publishing Company, Inc., Reading, 1968.

[30] M.E. Lesk and E. Schmidt, *Lex — a lexical analyzer generator*, Unix Programmer's Guide, Volume IIB, Bell Laboratories, Murray Hill, 1979.

[31] B. Liskov, et.al., *Abstraction mechanisms in CLU*, Communications of the ACM, 20, 8 (August 1977), pp. 564–576.

[32] R. Medina-Mora, *Syntax-directed Editing: Towards Integrated Programming Environments*, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, 1982.

[33] J.G. Mitchell, et.al., *Mesa Language Manual*, CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, April 1979.

[34] T.W. Reps, *Generating Language-Based Environments*, The MIT Press, Cambridge, 1984.

[35] S.P. Reiss, *PECAN: program development systems that support multiple views*, IEEE Transactions on Software Engineering, 11, 3 (March 1985), pp. 276–284.

[36] T.W. Reps and T. Teitelbaum, *The synthesizer generator*, ACM Software Engineering Notes, 9, 3 (May 1984), pp. 42–48.

[37] M.T. Rose, personal communication, June 1986.

[38] E. Sandewall, *Programming in an interactive environment: the 'Lisp' experience*, ACM Computing Surveys, 10, 1 (March 1978), pp. 35–71.

[39] Smith, David Andrew, *Rapid Software Prototyping*, Ph.D. dissertation, Computer Science Department, University of California, Irvine, May 12, 1982.

[40] G.L. Steele, *Common Lisp: the Language*, Digital Press, Maynard 1985.

[41] T.A. Standish and R.N. Taylor, *Arcturus: a prototype advanced Ada programming environment*, ACM Software Engineering Notes, 9, 3 (May 1984), pp. 57–64.

[42] S.L. Squires, M. Zelkowitz, M. Branstad, *Rapid prototyping workshop: overview*, ACM Software Engineering Notes, 7, 5 (December 1982), page 2.

[43] M.M. Syslo, *Algorithm 459 — the elementary circuits of a graph*, Collected Algorithms from CACM, Association for Computing Machinery, Inc., New York, 1978.

[44] F.P. Tadman, *The Arcturus Programming Environment Program Design and Rapid Prototyping Language*, Technical Report (unnumbered), Programming Environment Project, Computer Science Department, University of California, Irvine, October 1982.

[45] W. Teitelman, *InterLisp Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, 1975.

[46] W. Teitelman, *A tour through Cedar*, IEEE Transactions of Software Engineering, 11, 3 (March 1985), pp. 285–301.

[47] W. Teitelman and J. Mashley, *The Interlisp programming environment*, IEEE Computer, 14, 4 (April 1981), pp. 25–33.

[48] T .Teitelbaum and T.W. Reps, *The Cornell program synthesizer: a syntax-directed programming environment*, Communications of the ACM, 24, 9 (September 1981), pp. 563–573.

[49] H. Weinblatt, *A new search algorithm for finding the simple cycles of a finite directed graph*, Journal of the Association for Computing Machinery, 19, 1 (January 1972), pp. 43–56.

[50] M. Weisner, *Scale models and rapid prototyping*, ACM Software Engineering Notes, 7, 5 (December 1982), pp. 181–185.

[51] S.H. Willson, *The Arcturus User's Guide*, Technical Report (unnumbered), Programming Environment Project, Computer Science Department, University of California, Irvine, March 1983.

[52] S.H. Willson, *The Surveyor Programming Environment Reference Guide*, Technical Report 86-9R, Northrop Research and Technology Center, Palos Verdes, May 1986.

[53] S.H. Willson, *An Interactive Ada Interface for Solid Modeling*, Technical Report 86-10R, Northrop Research and Technology Center, Palos Verdes, May 1986.