A History of CLU

Barbara Liskov

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
(617)253-5886
liskov@lcs.mit.edu

April l992

Table of Contents

Abstract

The idea of a data abstraction has had a significant impact on the development of programming languages and on programming methodology. CLU was the first implemented programming language to provide direct linguistic support for data abstraction. This paper provides a history of data abstraction and CLU. CLU contains a number of other interesting and influential features, including its exception handling mechanism, its iterators, and its parameterized types.[1]

1. Introduction

The idea of a data abstraction arose from work on programming methodology. It has had a significant impact on the way modern software systems are designed and organized and on the features that are provided in modern programming languages. In the early and mid l970's, it led to the development of new programming languages, most notably CLU and Alphard. These language designs were undertaken to flesh out the idea and to provide direct support for new techniques for developing software.

This paper provides a history of CLU and data abstraction. CLU provides linguistic support for data abstraction; it was the first implemented language to do so. In addition, it contains a number of other interesting and influential features, including its exception handling mechanism, its iterators, and its parameterized types.

---

[1]This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, in part by the National Science Foundation under Grant CCR-8822158, and in part by the NEC Corporation of Tokyo, Japan.

The paper is organized as follows. Section 2 describes the work that led to the concept of data abstraction and how this concept came into existence. It also describes the beginning of the work on CLU, and discusses some of the later work on programming methodology that was based on data abstraction. Section 3 provides a history of the CLU development process, with emphasis on the design issues related to the technical features of CLU; the section contains material about related work in other languages as well. The paper concludes with an evaluation of CLU and a discussion of its influence on programming languages and methodology.

2. Data Abstraction

In my early work on data abstraction, in the latter part of 1972 through the summer of l973 , I was concerned with figuring out what the concept was, rather than designing a full programming language. This section traces the development of the concept and describes the environment in which the work occurred and the related work that was going on at that time.

A data abstraction, or abstract data type, is a set of objects and operations. Programs that access the objects can do so only by calling the operations, which provide means to observe an object's state and to modify it. The objects contain within them a storage representation that is used to store their state, but this representation is encapsulated: it is not visible to programs that use the data abstraction. For example, a set of integers type might have operations to create an empty set, to insert and delete elements from a set, to determine the cardinality of a set, and to determine whether a particular integer is a member of a set. A set might be represented by an array or a linked list, but since users can interact with it only by calling operations, the particular representation is not visible. One important benefit of such encapsulation is that it allows decisions about implementation to be changed without any need to modify programs that use the data abstraction.

The idea of a data abstraction developed in the early seventies. It grew out of work on programming methodology. At that time, there was a great deal of interest in methods for improving the efficiency of the programming process and also the quality of the product. There were two main approaches: structured programming, and modularity. Structured programming [Dijkstra, 1969] was concerned with program correctness (or reliability, as it was called in those days):

> The goal of structured programming is to produce program structures which are amenable to proofs of correctness. The proof of a structured program is broken down into proofs of the correctness of each of the components. Before a component is coded, a specification exists explaining its input and output and the function which it is supposed to perform [Liskov, 1972a, p. 193].

Not using gotos [Dijkstra, 1968a] was a part of structured programming because the resulting program structures were easier to reason about, but the idea of reasoning about the program at one level using specifications of lower level components was much more fundamental. The notion of stepwise refinement as an approach to constructing programs was also a part of this movement [Wirth, 1971].

The work on modularity [Liskov, 1972a; Parnas, 1971; Parnas, 1972a; Randell, 1969] was concerned with what program components should be like. For example, I proposed the idea of *partitions*:

> The system is divided into a hierarchy of partitions, where each partition represents one level of abstraction, and consists of one or more functions which share common resources. . . . The connections in data between partitions are limited to the explicit arguments passed from the functions of one partition to the (external) functions of another partition. Implicit interaction on common data may only occur among functions within a partition [Liskov, 1972a, p. 195].

This notion of partitions was based on Dijkstra's ideas about levels of abstraction [Dijkstra, 1968b] and my own work on the Venus operating system [Liskov, 1972b]. Venus was organized as a collection of partitions, each with externally accessible functions and hidden state information, which communicated by calling one another's functions.

The papers on programming methodology were concerned with system structure rather than with programming language mechanisms. They provided guidelines that programmers could use to organize programs but did not describe any programming language constructs that would help in the organization. The work on data abstraction arose from an effort to bridge this gap. Data abstraction merged the ideas of modularity and encapsulation with programming languages by relating encapsulated modules to data types. As a result, programming languages that provided direct support for modular programming came into existence, and a much clearer notion of what a module is emerged.

By the fall of l972, I had become dissatisfied with the papers on programming methodology, including my own, because I believed it was hard for readers to apply the notions to their own programs. The idea of a module was somewhat nebulous in these papers (some operations with hidden state information). Even less obvious was how to do modular design. The designer was supposed to identify modules, but it was not at all clear how this was to be done, and the papers provided relatively little guidance on this crucial point.

I noticed that many of the modules discussed in the papers on modularity were defining data types. For example, I had suggested that designers look for abstractions that hid the details of interacting with various hardware resources, and that hid the storage details of data structures [Liskov, 1972a]. This led me to think of linking modules to data types and eventually to the idea of abstract types with an encapsulated representation and operations that could be used to access and manipulate the objects. I thought that programmers would have an easier time doing modular design in terms of abstract types (instead of just looking for modules) because this was similar to deciding about data structures, and also because the notion of an abstract type could be defined precisely. I referred to the types as "abstract" because they aren't provided directly by a programming language but instead must be implemented by the user. An abstract type is abstract in the same way that a procedure is an abstract operation.

I gave a talk [Liskov, 1973a] on abstract data types at a workshop on Programming Languages and Operating Systems held in Savannah on April 9-12, l973. This talk merged the ideas of structured programming and modularity by relating the components of a structured program to either abstract operations (implemented by procedures) or abstract types. An abstract type provides

> a group of related functions whose joint actions completely define the abstraction as far as its users are concerned. The irrelevant details of how the abstraction is supported are completely hidden from the users [Liskov, 1973a, p. 6].

Furthermore, the language should support a syntactic unit that can be used to implement abstract types, and "the language must be strongly typed so that it is not possible to make use of an object in any other way [Liskov, 1973a, p. 7]" except by calling its operations.

At about the time of the Savannah meeting, I began to work with Steve Zilles, who was also at M.I.T. working on a similar idea. Steve published his ideas at Savannah [Zilles, 1973], and there were a number of other related talks given there, including a talk on monitors by Mike McKeag [McKeag, 1973] and a talk on Lis by Jean Ichbiah [Ichbiah, 1973].

Steve and I worked on refining the concept of abstract types over the spring and summer of 1973; Austin Henderson was involved to a lesser extent as an interested listener and critic. Our progress report for that year [Liskov, 1973b] describes a slightly later status for the work than what was reported at the Savannah meeting. In the progress report, we state that an abstract type should be defined by a "function cluster" containing the operations of the type. By the end of the summer, our ideas about language support for data abstraction were quite well-established, and Steve and I described them in a paper published in September [Liskov, 1973c]; a slightly later version of this paper appeared in the conference on very high level languages held in April 1974 [Liskov, 1974a]. The September paper states that an abstract data type is implemented by a "cluster" containing a description of the representation and implementations of all the operations. It defines structured programming:

> In structured programming, a problem is solved by means of a process of successive decomposition. The first step is to write a program which solves the problem but which runs on an abstract machine, i.e., one which provides just those data objects and operations which are suitable to solving the problem. Some or all of those data objects and operations are truly abstract, i.e., not present as primitives in the programming language being used [Liskov, 1973c, p. 3].

CLU was chosen as the name of the language in the fall of 1973. The name was selected because it is the first three letters of "cluster."

2.1. Related Early Work

Programming languages that existed when the concept of data abstraction arose did not support abstract data types, but some languages contained constructs that were precursors of this notion. (An analysis of language support for other languages was done by Jack Aiello, a student in the CLU group, in the fall of l973 and the spring of l974 [Aiello, 1974].) The mechanism that matched the best was the class mechanism of Simula 67. A Simula class groups a set of procedures with some variables. A class can be instantiated to provide an object containing its own copies of the variables; the class contains code that initializes these variables at instantiation time. However, Simula classes did not enforce encapsulation (although Palme later proposed a change to Simula that did [Palme, 1973]), and Simula was lacking several other features needed to support data abstraction, as discussed further in Section 3.2.

Extensible languages contained a weak notion of data abstraction. This work arose from a notion of "uniform referents" [Balzer, 1967; Earley, 1971; Ross, 1970]. The idea was that all data types ought to be referenced in the same way so that decisions about data representation could be delayed and changed. This led to a notion of a fixed set of operations that every type supported. For example, every type in EL1 [Wegbreit, 1972; Wegbreit, 1973] was permitted to provide five operations (conversion, assignment, selection, printing, and generation). However, the new type was not abstract; instead it was just an abbreviation for the chosen representation, which could be accessed everywhere, so that there was no encapsulation.

PL/I provided multi-entry procedures, which can be used to implement data abstractions, and in fact I have used PL/I in teaching how to program using abstract data types when languages with more direct support were not available. The PL/I mechanism allows the description of the representation chosen for objects of the new type to be grouped with the code for the operations; the representation is defined at the beginning of the multi-entry procedure, and the entry points serve as the operations. However, users of the type can access the representations of objects directly (without calling the operations), so again there is no enforcement of encapsulation. Furthermore, multi-entry procedures have other peculiarities, e.g., if control falls off the end of one entry point, it does not return to the caller but instead continues in the entry point that follows textually within the multi-entry procedure.

An interesting paper that appeared in early 1973 was Jim Morris' paper on protection in programming languages [Morris, 1973a]. This paper contains an example of an abstract data type implemented by a lambda expression that returns a list of operations to be used to access and manipulate the new object. The paper also describes how encapsulation can be enforced dynamically by means of a key that is needed to access the representation and whose value is known only to the type's operations (this notion is elaborated in [Morris, 1973b]). In the early design of CLU, we thought that this kind of dynamic checking would be needed in some cases, but as the design progressed we came to realize that complete static type checking is possible.

Bill Wulf and Mary Shaw published a paper concerning the misuse of global variables [Wulf, 1973] in l973. One point made in this paper is that there is no way in a block structured language to limit access to a group of variables to just the group of procedures that need such access. This paper represents some of the early thinking of the people who went on to develop Alphard [Shaw, 1981; Wulf, 1976].

Also in l973 Tony Hoare published an important paper about how to reason about the correctness of a data type implementation [Hoare, 1972]. This paper pointed the way to future work on specification and verification of programs built with data abstractions.

A timely and useful meeting was organized by Jack Dennis and held at the Harvard Faculty Club on October 3 - 5, 1973 to discuss issues in programming language design; a list of attendees is contained in Appendix A. The topics discussed were: types, semantic bases, concurrency, error handling, symbol identification, sharing and assignment, relation to systems. Most attendees at the meeting gave brief talks describing their research. I spoke about clusters and the current state of the CLU design, and Bill Wulf discussed the work on Alphard. (Neither of the language names existed at this point.) Ole-Johan Dahl discussed Simula classes and their relationship to clusters. Steve Zilles described his early work on specifying abstract data types; he gave the axioms for the soon to be notorious stack example. Carl

Hewitt discussed his work on Actors, and Tony Hoare described monitors. Also, Jim Mitchell described his early work on error handling, which led to the exception handling mechanism in Mesa [Mitchell, 1978].

2.2. Programming Methodology

The identification of data abstractions as an organizing principle for programs spurred work in programming methodology. This work is discussed briefly in this section.

As mentioned, the concept of data abstraction arose out of work on structured programming and modularity that was aimed at a new way of organizing programs. Traditionally, programs had been organized using procedures or subroutines. The new idea was to organize around modules that consisted of a number of related procedures, and, with the advent of data abstraction, these modules defined data types or objects. The hope was that this would lead to better organized programs that would be easier to implement and understand, and, as a result, easier to get right.

The resulting programming methodology [Liskov, 1979a; Liskov, 1986a] is object-oriented: programs are developed by thinking about the objects they manipulate and then inventing a modular structure based on these objects. Each type of object is implemented by its own program module. Although no studies have shown convincingly that this methodology is superior to others, the methodology has become widespread and people believe that it works. (I believe this; I also believe that it is impossible to run a controlled experiment that will produce a convincing result.)

A keystone of the methodology is its focus on independence of modules. The goal is to be able to deal with each module separately: a module can be implemented independently of (the code of) others, it can be reasoned about independently, and it can be replaced by another module implementing the same abstraction, without requiring any changes in the other modules of the program. Thus, for example, if a data abstraction were implemented too inefficiently, it could be reimplemented and the result would be a program that ran better, but whose behavior was otherwise unchanged. What's really interesting is that client programs don't have to be changed, and yet will run better.

Achieving independence requires two things: encapsulation and specification. Encapsulation is needed because if any other module depends on implementation details of the module being replaced, it will not be possible to do the replacement without changing that other module. Just having code in some other module that accesses (but does not modify) an object's representation is enough to make replacement impossible, since that code is dependent on implementation details. To keep programmers from writing such code, Dave Parnas advocated hiding code from programmers of other modules so that they would be unable to write code that depended on the details [Parnas, 1971]. I believe this position is too extreme, since it conflicts with other desirable activities such as code reading. Encapsulation makes it safer for programmers to read code.

Specifications are needed to describe what the module is supposed to do in an implementation-independent way so that many different implementations are allowed. (Code is not a satisfactory description since it doesn't distinguish what is required from ways of achieving it. One of the striking aspects of much of the work on object-oriented programming has been its lack of understanding of the importance of specifications; instead the code is taken as the definition of behavior.) Given a specification, one can develop an implementation without needing to consider any other part of the program. Furthermore, the program will continue to work properly when a new implementation is substituted for the old providing the new implementation satisfies the specification (and assuming the old implementation did too). Specifications also allow code that uses the abstraction to be written before code that implements the abstraction, and therefore are necessary if you want to do top-down implementation.

Work on specifications of data abstractions and on the related area of reasoning about correctness in programs that use and implement data abstraction started early [Hoare, 1972; Parnas, 1972b; Zilles, 1974a; Zilles, 1975] and continued for many years (see, e.g., [Wulf, 1976; Goguen, 1975; Guttag, 1975; Guttag, 1977; Berzins, 1979; Parnas, 1972b; Spitzen, 1975; Guttag, 1980]). Verification methods work only when the type's implementation is encapsulated (actually, protection against modification of the representation from outside the module is sufficient), since otherwise it will not be possible to limit one's attention to just a single module's implementation. If a language enforces encapsulation, independent reasoning about modules is on a sound foundation. Otherwise, it isn't and a complete proof requires a global analysis. In essence, having a language enforce encapsulation means that the compiler proves a global property of a program; given this proof, the rest of the reasoning can be localized.

Languages that enforce encapsulation are based on a "less is more" kind of philosophy. The idea is that something can be gained by having a programmer give up some freedom. What is gained is global: increased ease in reasoning about an entire, multi-module program. What is lost is local: a programmer must live by the rules, which can sometimes be inconvenient.

3. CLU

Although I was not thinking about developing a complete, implemented programming language when I first worked on data abstraction, I took this next step quite soon, sometime in the spring or summer of 1973. By the time work began in earnest on the language design in the fall of l973, many details of the language were already set. For example, we had already decided to implement abstract types with clusters, to keep objects in the heap, and to do complete type checking. However, there was lots of work left to do to design all the language features and their interactions.

In this section I provide a history of the CLU development. I begin by describing our goals (Section 3.1), the design process (Section 3.2), and our design principles (Section 3.3). The remaining sections discuss what I consider to be the important technical decisions made during the project. Details about project staffing and the phases of the project can be found in Appendices B and C contains respectively.

3.1. Language Goals

The primary goal of the project was to do research on programming methodology:

> We believe the best approach to developing a methodology that will serve as a practical tool for program construction is through the design of a programming language such that problem solutions developed using the methodology are programs in the language. Several benefits accrue from this approach. First, since designs produced using the methodology are actual programs, the problems of mapping designs into programs do not require independent treatment. Secondly, completeness and precision of the language will be reflected in a methodology that is similarly complete and precise. Finally, the language provides a good vehicle for explaining the methodology to others [Liskov, 1974b, p. 35].

We recognized early on that implementations are not the same as abstractions. An implementation is a piece of code; an abstraction is a desired behavior, which should be described independently from the code, by means of a specification. Thus, our original proposal to NSF says: "An abstract data type is a concept whose meaning is captured in a set of specifications, while a cluster provides an implementation of a data type [Dennis, 1974, p. 21]." An implementation is correct if it "satisfies" the abstraction's specification. There was a great deal of interest in the group in specification and verification techniques, especially for data abstractions. This started with Steve Zilles' early work on algebraic specifications, which was mentioned in the proposal and also in the progress report for l973-4 [Liskov, 1974b]; work on specifications is discussed briefly in Section 2.2. However, unlike the Alphard group, we chose to separate the work on specifications from the language definition. I believed that the language should contain only declarations that the compiler could make use of, and not statements that it would treat simply as comments. I think this decision was an important factor in our ability to make quick progress on the language design.

The work on CLU occurred at MIT within the Laboratory for Computer Science with support from the National Science Foundation and DARPA. We believed that our main "product" was concepts rather than the language. We thought of our primary output as being publications, and that success would be measured by our influence on programming methodology and practice and on future programming languages. We did not think of CLU as a language that would be exported widely. Instead, we were concerned primarily with the export of ideas.

CLU was intended to be a general purpose programming language that supports most applications, although it was geared more toward symbolic than numerical computing. It was not oriented toward low-level system programming (e.g., of operating systems), but it can be used for this purpose by the addition of a few data types implemented in assembler, and the introduction of some procedures that provide "unsafe features." This is the technique that we have used in our implementations. For example, we have a procedure in the library called "_cvt" that can be used to change the type of an object. Such features are not described in the CLU reference manual; most users are not supposed to use them. I believe this is a better approach than providing a generally unsafe language like C, or a language with unsafe features, like Mesa [Mitchell, 1978], since it discourages programmers from using the unsafe features casually.

CLU was intended to be used by "experienced" programmers. Programmers need not be wizards, but they aren't supposed to be novices either. Although CLU (unlike Pascal) was not designed to be a teaching language, we use it this way and it seems to be easy to learn (we teach it to sophomores).

CLU was geared toward developing production code. It was intended to be a tool for "programming in the large," for building big systems (e.g., several hundred thousand lines) that require many programmers to work on them. As the work on CLU went on, I developed a programming methodology for such systems [Liskov, 1979a; Liskov, 1986a]. CLU favors program readability and understandability over ease of writing, since we believed that these were more important for our intended users.

3.2. The Design Process

There were four main language designers: myself, and three graduate students, Russ Atkinson, Craig Schaffert, and Alan Snyder. Steve Zilles was deeply involved in the early work on the language, but by 1974 Steve was concentrating primarily on his work on specifications of abstract types, and acted more as an interested onlooker and critic of the developing design. As time went by, other students joined the group including Bob Scheifler and Eliot Moss. (A list of those who participated in the design is given in Appendix B.)

The design was a real group effort. Usually it is not possible to identify an individual with a feature (iterators are the exception here, as discussed in Section 3.10). Instead ideas were developed in meetings, worked up by individuals, evaluated in later meetings, and then reworked.

I was the leader of the project and ultimately made all the decisions, although often we were able to arrive at a consensus. In our design meetings we sometimes voted on alternatives, but these votes were never binding. I made the actual decisions later.

Russ, Craig, and Alan (and later Bob and Eliot) were implementers as well as designers. All of us acted as "users;" we evaluated every proposal from this perspective (considering its usability and expressive power), as well as from a designer's perspective (considering both implementability, and completeness and well-definedness of semantics).

We worked on the implementation in parallel with the design. We did not allow the implementation to define the language, however. We delayed implementing features until we believed that we had completed their design, and if problems were discovered, they were resolved in design meetings. Usually, we did not introduce any new features into the language during the implementation, but there were a few exceptions (in particular, own data).

We provided external documentation for the language through papers, reference manuals, and progress reports. We documented the design as it developed in a series of internal design notes [PMG, 1979a]. There were seventy-eight notes in all; the first was published on December 6, 1973 and the last on July 30, 1979. The notes concentrated on the utility and semantics of proposed language features. Typically, a note would describe the meaning of a feature (in English) and illustrate its use through examples. Syntax was introduced so that we could write code fragments, but was always considered to be of secondary importance. We tended to firm up syntax last, at the end of a design cycle.

The group held weekly design meetings. In these meetings we evaluated proposed features as thoroughly as we could. The goal was to uncover any flaws, both with respect to usability and semantics. This process seemed to work well for us: we had very few surprises during implementation. We published (internal) design meeting minutes for most of our meetings [PMG, 1979b].

The design notes use English to define the semantics of proposed constructs. We did not use formal semantics as a design tool because I believed that the effort required to write the formal definitions of all the many variations we were considering would greatly outweigh any benefit. We relied on our very explicit design notes and thorough analysis instead. I believe our approach was wise and would recommend it to designers today. During design what is needed is precision, which can be achieved by doing a careful and rigorous, but informal, analysis of semantics as you go along. It's the analysis process that is important; in its absence, a formal semantics is probably not much help. We provided a formal semantics (in several forms) when the design was complete [Schaffert, 1978; Scheifler, 1978]. It validated our design but did not uncover errors, which was gratifying. For us, the main virtue of the formal definition was as *ex post facto* documentation.

The group as a whole was quite knowledgeable about languages that existed at the time. I had used Lisp extensively and had also programmed in Fortran and Algol 60, Steve Zilles and Craig Schaffert had worked on PL/I compilers, and Alan Snyder had done extensive programming in C. In addition, we were familiar with Algol 68, EL/1, Simula 67, Pascal, SETL, and various machine languages. Early in the design process we did a study of other languages to see whether we should use one of them as a basis for our work [Aiello, 1974]. We ultimately decided that none would be suitable as a basis. None of them supported data abstraction, and we wanted to see where that idea would lead us without having to worry about how it might interact with pre-existing features. However, we did borrow from existing languages. Our semantic model is largely borrowed from Lisp; our syntax is Algol-like.

We also had certain negative influences. We felt that Pascal had made too many compromises to simplify its implementation. We believed strongly in compile-time type checking but felt it was important for the language to support types in a way that provided adequate expressive power. We thought Pascal was deficient here, e.g., in its inability (at the time) to support a procedure that could accept as an argument arrays of different sizes. We felt that Algol 68 had gone much too far in its support for overloading and coercions. We believed that a language must have very simple rules in this area or programs would be hard for readers to understand. This led us ultimately to our ideas about "syntactic sugar" (see Section 3.7).

Simula 67 was the existing language that was closest to what we wanted, but it was deficient in several ways, some of which seemed difficult to correct:

1. Simula did not support encapsulation, so its classes could be used as a data abstraction mechanism only if programmers obeyed rules not enforced by the language.
2. Simula did not provide support for user-defined type "generators." These are modules that define groups of related types, e.g., a user-defined set module that defines set[int], set[real], etc.
3. It did not group operations and objects in the way we thought they should be grouped, as discussed in Section 3.4.
4. It treated built-in and user-defined types non-uniformly. Objects of user-defined types had to reside in the heap, but objects of built-in type could be in either the stack or the heap.

In addition, we felt that Simula's inheritance mechanism was a distraction from what we were trying to do. Of course, this very mechanism was the basis for another main language advance of the seventies, Smalltalk. The work on Smalltalk was concurrent with ours and was completely unknown to us until around 1976.

3.3. Design Principles

The design of CLU was guided by a number of design principles, which were applied quite consciously. The principles we used were the following:

1. Keep focussed. The goal of the language design was to explore data abstraction and other mechanisms that supported our programming methodology. Language features that were not related to this goal were not investigated. For example, we did not look at extensible control structures. Also, although I originally intended CLU to support concurrency [Dennis, 1974], we focussed on sequential programs initially to limit the scope of the project, and eventually decided to ignore concurrency entirely. (We did treat concurrency in a successor language, Argus [Liskov, 1983; Liskov, 1988].)

2. Minimality. We included as few features as possible. We believed that we could learn more about the need for a feature we were unsure of by leaving it out: if it was there, users would use it without thinking about whether they needed it, but if it was missing, and they really needed it, they would complain.

3. Simplicity. Each feature was as simple as we could make it. We measured simplicity by ease of explanation; a construct was simple if we could explain it (and its interaction with other features) easily.

4. Expressive power. We wanted to make it easy for users to say the things we thought they needed to say. This was a major motivation, for example, for the exception mechanism. To a lesser extent, we wanted to make it hard to express things we thought should not be expressed, but we didn't pay too much attention to this; we knew that users could write (what we thought were) bad programs in CLU if they really wanted to.

5. Uniformity. As much as possible, we wanted to treat built-in and user-defined types the same. For example, operations are called in the same way in both cases; user-defined types can make use of infix operators, and built-in types use our "type_name$op_name" syntax to name operations that do not correspond to operator symbols just like the user-defined types.

6. Safety. We wanted to make it as easy as possible for programmers to write correct code by ruling out errors or making it possible to detect them automatically. This is why we have strong type checking, a garbage collected heap, and bounds checking.

7. Performance. We wanted CLU programs to run quickly, e.g., close to comparable C programs. We also wanted fast compilation but this was of secondary importance.

As usual, several of these goals are in conflict. Expressive power conflicts with minimality; performance conflicts with safety and simplicity. When conflicts arose we resolved them as best we could, by trading off what was lost and what was gained in following a particular approach. For example, we based our semantics on a garbage collected heap even though it may require more expense at runtime because it improves program safety and simplifies our data abstraction mechanism. A second example is our iterator mechanism; we limited the expressive power of the mechanism so that we could implement it using a single stack.

Concern for performance pervaded the design process. We always considered how proposed features could be implemented efficiently. In addition, we expected to make use of compiler optimizations to improve performance. Programming with abstractions means there are lots of procedure calls, e.g., to invoke the operations of abstract types. In our September l973 paper, Steve and I noted:

> The primary business of a programmer is to build a program with a good logical structure -- one which is understandable and leads to ease in modification and maintenance. . . . We believe it is the business of the compiler to map good logical structure into good physical structure. . . . Each operator-use may be replaced either by a call upon the corresponding function in the cluster or by inline code for the corresponding function. . . . Inline insertion of the code for a function allows that code to be subject to the optimization transformations available in the compiler [Liskov, 1973c, p. 32-33].

Thus we had in mind inline substitution followed by other optimizations. Bob Scheifler did a study of how and when to do inline substitution for his BS thesis [Scheifler, 1976; Scheifler, 1977].

3.4. Implementing Abstract Types

In CLU an abstract data type is implemented by a *cluster*, which is a program module with three parts (see Figure 3-1): (1) a header listing the operations that can be used to create and interact with objects of that type; (2) a definition of the storage representation, or *rep*, that is used to implement the objects of the type; (3) procedures (and iterators -- see Section 3.10) that implement the operations listed in the header (and possibly some additional, internal procedures and iterators as well). Only procedures inside the cluster can access the representations of objects of the type. This restriction is enforced by type checking.

```
int_set = cluster is create, member, size, insert, delete, elements

    rep = array[int]

    % implementations of operations go here

    end int_set
```

Figure 3-1: The Structure of a Cluster

There are two different ways to relate objects and operations:

1. One possibility is to consider the operations as belonging to the type. This is the view taken by both CLU and Alphard. (Alphard "forms" are similar to clusters.) In this case, a type can be thought of as defining both a set of objects and a set of operations. (The approach in Ada is a variation on this. A single module defines both the type and the operations. Operations have access to the representation of objects of their type because they are defined inside the module that defines the type. Several types can be defined in the same modules, and operations in the module can access the representations of objects of all these types.)

2. A second possibility is to consider the operations as belonging to the objects. This is the view taken in Simula, and also in Smalltalk and C++.

These two approaches have different strengths and weaknesses. The "operations in type" approach works well for operations like "+" or "union" that need to access the representations of two or more objects at once, since with this approach, any operation of the type can access the representation of any object of the type. The "operations in object" approach does not work so well for such operations because an operation can only access the representation of a single object, the one it belongs to. On the other hand, if there can be more than one implementation for a type in existence within a program, or if there is inheritance, the "operations in object" approach works better, since an operation knows the representation of its object, and cannot make possibly erroneous assumptions about the representation of other objects since it is unable to access these representations.

We believed that it was important to support multiple implementations but even more important to make binary operations like "+" work well. We did not see how to make the "operations in object" approach run efficiently for binary operations. For example, we wanted adding two integers to require a small number of machine instructions, but this is not possible unless the compiler knows the representation of integers. We could have solved this problem by treating integers specially (allowing just one implementation for them), but that seemed inelegant, and it conflicted with our uniformity goal. Therefore, a program in CLU can have only a single implementation for any given type (built-in or user-defined).

People have been working for the last fifteen years to make integers run fast in object-oriented languages (see, e.g., the work of Dave Ungar and Craig Chambers [Chambers, 1990]). So in retrospect, it was probably just as well that we avoided this problem. Furthermore, although a program development support system must store many implementations of a type (and we do this as discussed in Section 3.11), allowing multiple implementations within a single program seems less important. During the design of CLU, we hypothesized that it might be sufficient to limit different implementations to different regions of a program [Liskov, 1975a]. This idea was incorporated into Argus [Liskov, 1983; Liskov, 1988], the language we developed after CLU. In Argus the regions are called "guardians" and the same type can have different implementations in different guardians within the same program. Guardians can communicate using objects of a type where the implementations differ because the representation can change as part of the communication [Herlihy, 1982].

3.5. Semantic Model

CLU looks like an Algol-like language, but its semantics is like that of Lisp: CLU objects reside in an object universe (or heap), and a variable just identifies (or refers to) an object. We decided early on to have objects in the heap, although we had numerous discussions about the cost of garbage collection. This decision greatly simplified the data abstraction mechanism (although I don't think we appreciated the full extent of the simplification until quite late in the project). A language that allocates objects only on the stack is not sufficiently expressive; the heap is needed for objects whose sizes must change and for objects whose lifetime exceeds that of the procedure that creates them. (Of course, it's possible for a program to do everything using a stack — or common — but only at the cost of doing violence to the program structure.) Therefore, the choice is: just heap, or both.

Here are the reasons why we chose the heap approach (an expanded discussion of these issues can be found in [Moss, 1978]):

1. Declarations are simple to process when objects are on the heap: the compiler just allocates space for a pointer. When objects are on the stack, the compiler must allocate enough space so that the new variable will be big enough to hold the object. The problem is that knowledge about object size ought to be encapsulated inside the code that implements the object's type. There are a number of ways to proceed. For example, Ada requires that size information be made available to the compiler; this is why type definitions appear in the "public" part of a module. However, this means that the type's representation must be defined before any modules that use the type can be compiled, and also, if the representation changes, all using modules must be recompiled. In Alphard, the plan was to provide additional operations (not available to user code) that could be called to determine how much space to allocate; the compiler would insert calls to these operations when allocating space for variables. The important point is that with the heap approach the entire problem is avoided.

2. The heap approach allows us to separate variable and object creation: variables are created by declarations, while objects are created explicitly by calling an operation. The operation can take arguments if necessary and can ensure that the new object is properly initialized so that it has a legitimate state. (In other words, the code can insure that the new object satisfies the rep invariant [Liskov, 1986b].) In this way we can avoid a source of program errors. Proper initialization is more difficult with the stack approach since arguments are often needed to do it right. Also, the heap approach allows many creation operations, e.g., to create an empty set, or a singleton set; having many creation operations is more difficult with the stack approach.

3. The heap approach allows variable and object lifetimes to be different; with the stack approach they must be the same. If the object is to live longer than the procedure that creates it, a global variable must be used. With the heap approach, a local variable is fine; later a variable with a longer lifetime can be used, e.g., in the calling procedure. Avoiding global variables is good since they interfere with the modular structure of the program (as was pointed out by Wulf and Shaw [Wulf, 1973]).

4. Assignment has a type-independent meaning with the heap approach;

   ```
   x := e
   ```

   causes x to refer to the object obtained by evaluating expression e. With the stack approach, evaluating an expression produces a value that must be copied into the assigned variable. To do this right requires a call on the object's assignment operation, so that the

> right information is copied. In particular, if the object being copied contains pointers, it is not clear whether to copy the objects pointed at; only the implementer of the type knows the right answer. The assignment operation needs access to the variable containing the object being copied; really call-by-reference is required. Alphard did copying right since it allowed the definers of abstract types to define the assignment operation using by-reference parameters. Ada did not allow user-defined operations to control the meaning of assignment; at least in part this is because of the desire to treat call-by-reference as an optimization of call-by-value/result, which simply won't work in this case.

One unusual aspect of CLU is that our procedures have no free (global) variables (this is another early decision that is discussed in our September, l973 paper [Liskov, 1973c]). The view of procedures in CLU is similar to that in Lisp: CLU procedures are not nested (except that there can be local procedures within a cluster) but instead are defined at the "top" level, and can be called from any other module. In Lisp such procedures can have free variables that are scoped dynamically, a well-known source of confusion. We have found that free variables are rarely needed. This is probably attributable partly to data abstraction itself, since it encourages grouping related information into objects, which can then be passed as arguments.

In fact, CLU procedures do not share variables at all. In addition to there being no free variables, there is no call-by-reference. Instead arguments are passed "by object"; the (pointer to the) object resulting from evaluating the actual argument expression is assigned to the formal. (Thus passing a parameter is just doing an assignment to the formal.) Similarly, a pointer to a result object is returned to the caller. We have found that ruling out shared variables seems to make it easier to reason about programs.

A CLU procedure can have side effects only if the argument objects can be modified (since it cannot access the caller's variables). This led us to the concept of "mutable" objects. Every CLU object has a state. The states of some objects, such as integers and strings, cannot change; these objects are "immutable." Mutable objects (e.g., records and arrays) can have a succession of states. We spent quite a bit of time discussing whether we should limit CLU to just immutable objects (as in pure Lisp) but we concluded that mutable objects are important when you need to model entities from the real world, such as storage. (Probably this discussion would not have been so lengthy if we had not had so many advocates of dataflow attending our meetings!) It is easy to define a pure subset of CLU by simply eliminating the built-in mutable types (leaving their immutable counterparts, e.g., sequences are immutable arrays).

CLU assignment causes sharing: after executing "x := y," variables x and y both refer to the same object. If this object is immutable, programs cannot detect the sharing, but they can if the shared object is mutable, since a modification made via one variable will be visible via the other one. People sometimes argue that sharing of mutable objects makes reasoning about programs more difficult. This has not been our experience in using CLU. I believe this is true in large part because we do not manipulate pointers explicitly. Pointer manipulation is clearly both a nuisance and a source of errors in other languages.

The cost of using the heap is greatly reduced by keeping small immutable objects of built-in types, such as integers and booleans, directly in the variables that refer to them. These objects fit in the variables (they are no bigger than pointers) and storing them there is safe since they are immutable: Even though in this case assignment does a copy of the object, no program can detect it.

3.6. Issues Related to Safety

Our desire to make it easy to write correct programs led us to choose constructs that either ruled out certain errors entirely or made it possible to detect them automatically.

We chose to use garbage collection because certain subtle program errors aren't possible under this semantics. Explicit deallocation is unattractive from a correctness point of view, since it can lead to both dangling references and storage leaks; garbage collection rules out these errors. The decision to base CLU on a garbage-collected heap was made during the fall of l973 [Liskov, 1974a].

Another important effect of the safety goal was our decision to have static type checking. We included here both checking within a module (e.g., a procedure or a cluster) and inter-module type checking; the interaction of type checking with separate compilation is discussed in Section 3.11. Originally we thought we would need to do runtime checking [Liskov, 1973c] and we planned to base our technique on that of Morris [Morris, 1973a]. By early l974, we realized that we could do compile time checking most of the time [Liskov, 1974a]; this issue is discussed further in Section 3.7.

We preferred compile-time checking to runtime checking because it enabled better runtime performance and allowed us to find errors early. We based our checking on declarations, which we felt were useful as a way of documenting the programmer's intentions. (This position differs from that in ML [Milner, 1990], in which type information is inferred from the way variables are used. We were not aware of work on ML until the late seventies.) To make the checking as effective as possible, we ruled out coercions (automatic type conversions). We avoided all declarative information that could not be checked. For example, we discussed declaring within a cluster whether the type being implemented was immutable. We rejected this because the only way the compiler could ensure that this property held was to disallow mutable representations for immutable types. We wanted to allow an immutable type to have a mutable representation. One place where this is useful is to support "benevolent side effects" that modify the representation of an object to improve performance of future operation calls without affecting the visible behavior of the object.

Type checking in CLU uses both structure and name equality. Name equality comes from clusters. If "foo" and "bar" are the names of two different clusters, the two types are not equal. (This is true if they have the same representations; it is also true if they have the same operations with the same signatures.) Structure equality comes from "equates." For example, if we have the two equates

```
t = array[int]
s = array[int]
```

then t = s. We decided not to allow recursion in equates on the grounds that recursion can always be accomplished by using clusters. Although this reasoning is correct, the decision was probably a mistake; it makes certain programs awkward to write since extraneous clusters must be introduced just to get the desired recursion.

Another decision made to enhance safety was to not require that variables be initialized when they are declared. CLU allows declarations to appear anywhere; they are not limited to just the start of a block. Nevertheless, sometimes when a variable is declared there is no meaningful object to assign to it. If the language requires such an assignment, it misses a chance to notice automatically if the variable is used before it is assigned. The definition of CLU states that this situation will be recognized. It is recognized when running under the debugger, but the necessary checking has never been implemented by the compiler. This is the only thing in CLU that has not been implemented. Checking for proper variable initialization can usually be done by the compiler (using simple flow analysis), which would insert code to do runtime checks only for the few variables where the analysis is inconclusive. However, we never added the checking to the compiler (because of lack of manpower), and we did not want to do runtime checking at every variable use.

By contrast, we require that all parts of an object be initialized when the object is created, thus avoiding errors arising from missing components. We believed that meaningful values for all components exist when an object is created; in part this is true because we don't create the object until we need to, in part because creation happens as the result of an explicit call with arguments, if necessary, and in part because of the way CLU arrays are defined (see below). This belief has been borne out in practice.

The differing positions on variable and object component initialization arose from an evaluation of performance effects as well as from concerns about safety. As mentioned, checking for variable initialization can usually be done by the compiler. Checking that components are initialized properly is much more likely to need to be done at runtime.

Finally, we took care with the definitions of the built-in types both to rule out errors and to enable error detection. For example, we do bounds checking for ints and reals. Arrays are especially interesting in this regard. CLU arrays cannot have any uninitialized elements. When they are created, they contain some elements (usually none) provided as arguments to the creation operation; thereafter they can grow and shrink on either end; each time an array grows, the new element is supplied. Furthermore, bounds checking is done on every array operation that needs it (e.g., when the ith element is fetched, we check to be sure that i is a legal index in the array). Finally, arrays provide an iterator (see Section 3.10) that yields all elements and a second iterator that yields all legal indices, allowing a programmer to avoid indexing errors altogether.

3.7. Parametric Polymorphism

I mentioned earlier that we wanted to treat built-in and user-defined types alike. Since built-in types could make use of parameters, we wanted to allow them for user-defined types too. At the same time we wanted to provide complete type checking for them.

For example, CLU arrays are parameterized. An "array" is not a type by itself. Instead it is a "type generator" that, given a type as a parameter, produces a type. Thus, given the parameter int, it produces the type array[int]. We say that providing the parameter causes the type to be "instantiated." It is clearly useful to have parameterized user-defined types; e.g., using this mechanism we could define a set type generator that could be instantiated to provide set[int], set[char], set[set[char]], and so on.

The problem with type generators is how to type-check the instantiations. We limit actual values of parameters to compile time constants such as "3", "int", and "set[int]." However, when the parameter is a type, the type generator may need to use some of its operations. For example, to test for set membership requires the ability to compare objects of the parameter type for equality. Doing this requires the use of the equal operation for the parameter type.

Our original plan was to pass in a type-object (consisting of a group of operations) as an argument to a parameterized module, and have the code of the module check at runtime whether the type had the operations it needed with the proper signatures. Eventually we invented the "where" clause [Liskov, 1977a], which describes the names and signatures of any operations the parameter type must have, e.g.,

set = **cluster** [t: **type**] **is** create, member, size, insert, delete, elements
  where t **has** equal: **proctype** (t, t) **returns** (bool)

Inside the body of a parameterized module, the only operations of a type parameter that can be used are those listed in the where clause. Furthermore, when the type is instantiated, the compiler checks that the actual type parameter has the operations (and signatures) listed in the where clause. In this way, complete compile-time checking occurs.

CLU was way ahead of its time in its solution for parameterized modules. Even today, most languages do not support parametric polymorphism, although there is growing recognition of the need for it (e.g., [Cardelli, 1988]).

3.8. Other Uniformity Issues

In the previous section I discussed how user-defined types can be parameterized just like built-in ones. In this section I discuss two other uniformity issues, the syntax of operation calls, and syntax for expressions. I also discuss the way CLU views the built-in types, and what built-in types it provides.

A language like CLU that associates operations with types has a naming problem: many types will have operations of the same name (e.g., create, equal, size), and when an operation is called we need some way of indicating which one is meant. One possibility is to do this with overloading, e.g., "equal" denotes many procedures, each with a different signature, and the one intended is selected by considering the context of the call. This rule works fairly well (assuming no coercions) when the types of the arguments are sufficient to make the determination, e.g., equal(s, t) denotes the operation named "equal" whose first argument is of s's type and whose second argument is of t's type. It doesn't work so well if the calling context must be considered, which is the case for all creation operations. For example, we can tell that set create is meant in the following code:

s: set[int] := create( )

but it's more difficult (and sometimes impossible) if the call occurs within an expression.

We wanted a uniform rule that applied to all operations of a type, including creation operations. Also, we were vehemently opposed to using complicated rules to resolve overloading (e.g., as in Algol 68). This led us to require instead that every call indicate explicitly the exact operation being called, e.g.,

s: set[int] := set[int]$create( )

In doing so, we eliminated overloading all together: the name of an operation is always t$o, where t is the name of its type, and o is its name within its type. This rule is applied uniformly to both built-in and user-defined types.

We also allow certain short forms for calls. Most languages provide an expression syntax that allows symbols such as "+" to be used and allows the use of infix notation. We wanted to provide this too. To accomplish this we used Peter Landin's notion of "syntactic sugar" [Landin, 1964]. We allow common operator symbols but these are only short forms for what is really happening, namely a call on an operation using its full t$o name. When the compiler encounters such a symbol, it "desugars" it by following a simple rule: it produces t$o where t is the type of the first argument, and o is the operation name associated with the symbol. Thus "x + y" is desugared to "t$add(x, y)" where t is the type of x. Once the desugaring has happened, the compiler continues processing using the desugared form (it even does type checking using this form). In essence the desugared form is the canonical form for the program.

Not only is this approach simple and easy to understand, it applies to both built-in and user-defined types uniformly. To allow sugars to be used with a new type, the type definer need only choose the right names for the operations. For example, to allow the use of +, he or she names the addition operation "add." This notion of desugaring applies to all the arithmetic operations, to equality and related operations (e.g., <), and also to the operations that access and modify fields of records and elements of arrays.

We did not succeed in making built-in and user-defined types entirely alike, however. Some built-in types have literals (e.g., ints). Although we considered having a special literal notation for user-defined types, in the end we concluded that it offered very little advantage over regular calls of creation operations. Another difference is that there is more power in our parameterization mechanism for records than exists for user-defined types. In essence a record type generator is parameterized by the names and types of its fields; different instantiations can have different numbers of fields, and the operation names are determined by the field names. User-defined type generators must have a fixed number of parameters, and the operation names are fixed when the type generator is defined.

Nevertheless, we achieved a design with a high degree of uniformity. This ultimately colored our view of the built-in types. We ceased to think of them as something special; instead they were just the types we provided. This led us to decide that we need not be parsimonious with the built-in types. For example, all the type generators come in mutable/immutable pairs, e.g., array/sequence, record/struct, variant/oneof (these are tagged unions), although just providing the mutable generators would have been sufficient (and the decision to provide both mutable and immutable generators was made very late in the design). Naturally we thought of the built-in types in terms of their operations, since this was how we thought about all types. We were generous with the operations for built-in types: we provided all operations that we thought users might reasonably need, rather than a small subset that would have been semantically complete. I believe this is the proper view when defining a type (either built-in or user-defined) that is expected to be used by many different people.

The built-in types of CLU are similar to those of other modern languages. Procedures are first class values in CLU; we permit them to be passed as arguments, returned as results, and stored in data structures. We have an easy time with procedures because they are not allowed to have free variables and therefore we do not need to create closures for them. Recursive calls are permitted.

CLU provides a type called "any" which is the union of all types. An object of type any can be "forced" at runtime to its underlying type, but this does not lead to type errors, since an attempt to force the object to the wrong type will fail. A type like any is needed in a statically typed language; in essence it provides an escape to runtime type-checking.

3.9. Exception Handling

I have already discussed the fact that the main goal of the work on CLU was to support programming methodology. We had a strong belief that some kind of exception mechanism was needed for this. We wanted to support "'robust' or 'fault-tolerant' programs, i.e., programs that are prepared to cope with the presence of errors by attempting various error recovery techniques [Liskov, 1975b, p. 9]." This means they must be prepared to check for "exceptional" conditions and to cope with them when they occur; a majority of the code is often dedicated to this. Without a good mechanism this code is both hard to write and difficult to read. Also, we believed that support for exceptions

> strengthens the abstraction power of the language. Each procedure is expected to be defined over all possible values of its input parameters and all possible actions of the procedures it calls. However, it is not expected to behave in the same way in all cases. Instead, it may respond appropriately in each case [Liskov, 1975b, p. 11].

Therefore, we decided that CLU ought to have an exception mechanism. Support for such a mechanism was already a goal in early l974 [Zilles, 1974b]. In doing the design, we were aware of mechanisms in PL/I, Mesa [Mitchell, 1978; Lampson, 1974], and also Roy Levin's thesis [Levin, 1977] and the paper by John Goodenough [Goodenough, 1975].

CLU provides an exception mechanism based on the termination model of exceptions: A procedure call terminates in one of a number of conditions; one is the "normal" return and the others are "exceptional" terminations. We considered and rejected the resumption model present in both PL/I and Mesa because it was complex and also because we believed that most of the time, termination was what was wanted. Furthermore, if resumption were wanted, it could be simulated by passing a procedure as an argument (although closures would be useful here).

CLU's mechanism is unusual in its treatment of unhandled exceptions. Most mechanisms pass these through: if the caller does not handle an exception raised by a called procedure, the exception is propagated to its caller, and so on. We rejected this approach because it did not fit our ideas about modular program construction. We wanted to be able to call a procedure knowing just its specification, not its implementation. However, if exceptions are propagated automatically, a procedure may raise an exception not described in its specification.

Although we did not want to propagate exceptions automatically, we also did not want to require that the calling procedure handle all exceptions raised by the called procedure, since often these represented situations in which there was nothing the caller could do. For example, it would be a nuisance to have to provide handlers for exceptions that ought not to occur, such as a bounds exception for an array access when you have just checked that the index is legal. Therefore, we decided to turn all unhandled exceptions into a special exception called "failure" and propagate it. This mechanism seems to work well in practice.

The main decisions about our exception mechanism had been made by June l975 [Liskov, 1975b], but we noted that "The hardest part of designing an exception handling mechanism, once the basic principles are worked out, is to provide good human engineering for catching exceptions [Liskov, 1975b, p. 13]." We worked out these details over the following two years. We had completed the design by the fall of 1977; the mechanism is described in [Liskov, 1977b; Liskov, 1978a; Liskov, 1979b].

CLU exceptions are implemented efficiently [Liskov, 1978b]. As a result, they are used in CLU programs not just to indicate when errors occur but as a general way of conveying information from a called procedure to its caller.

3.10. Iterators

One of the tenets of the CLU design was that we were not going to do research in control structures. However, we did such work in defining the exception mechanism, and also in designing certain other control structures to make up for the lack of gotos (e.g., the **break** statement, which terminates a loop). We also did it in defining iterators.

Iterators were inspired by a construct in Alphard called a "generator" [Shaw, 1976; Shaw, 1977]. We first learned about this in the summer of 1975 when we visited the Alphard group at CMU. We were intrigued by generators since they solved some problems with data abstractions, but we thought they were too complicated. Russ Atkinson designed iterators on the airplane going back to Boston after this meeting and described them in a design note in September 1975 [Atkinson, 1975].

The problem solved by both generators and iterators is the following. Many data abstractions are collections of elements, and the reason for collecting the elements is so that later you can do something to them. Examples of such collections are arrays, sets, and lists. The problem is that for some types there is no obvious way to get to the elements. For arrays, you can use indexes; for lists, you can follow links. But for sets it's not clear what to do. What you would like is an operation of the type to provide the elements. Such an operation could be a procedure that returns an array containing the elements, but that is expensive if the collection is large. Instead, it would be nice to get at the elements one at a time. A generator does this by providing a group of operations, containing at least an operation to get the generation started, an operation to get the next element, and an operation to determine whether there are any more elements. Alphard generators had several more operations, and the Alphard designers worked out a way to use the **for** statement to call these operations at appropriate points.

A CLU iterator is a single operation that yields its results incrementally. For example,

elements = **iter** (s: set[t]) **yields** (t)

produces all the elements in set s, but it yields them one at a time. An iterator is called in a **for** statement:

for x: int **in** set[int]$elements(coll) **do**
...

The for loop begins by calling the iterator. Each time the iterator yields a result, the loop body is executed; when the body finishes, control resumes in the iterator, and when the iterator returns, the loop terminates. Also, if the loop body causes the loop to terminate, the iterator terminates.

Iterators are related to coroutines; the iterator and the body of the for loop pass control back and forth. However, their use is limited so that CLU programs can make do with a single stack. They are inexpensive: a yield effectively calls the loop body, which returns to the iterator when it is finished. (Calls are very cheap in CLU.) Imposing the limitations on iterators was done to get the efficient, single stack implementation, albeit at the expense of some expressive power. For example, iterators cannot be used to compute whether two lists have the same elements, since to do this you need to iterate through the two lists side-by-side, and CLU only allows iterators to be nested.

3.11. Putting Programs Together

From the start, we believed that modules should be compiled separately and linked together to form programs. Furthermore we wanted to be able to compile programs that used abstractions before the used abstractions had been implemented or even fully defined (in the case of an abstract type, only some of the type's operations may be known when the type is invented). Nevertheless, we wanted to have complete inter-module type checking, and we wanted the checking to be accurate: when compiling a using module, we wanted to check the actual interface of the used module rather than some local definitions that might be wrong. (CLU modules are procedures, clusters, and iterators.)

By September l973, we had already decided that of CLU programs should be developed within a program library [Liskov, 1973c]. The library contained "description units," each of which represented an abstraction. A description unit contained an interface specification for its abstraction; for a data abstraction, this consisted of the names and signatures of the operations. The description unit also contained zero or more implementations. Its interface specification could not be changed (after an initial period when the abstraction is being defined), but new implementations could be added over time. (The interface specification is similar to a Mesa "def module.")

When compiling a module, the compiler would use the interface specifications in description units in the library to type check all its uses of other modules. The module uses local names to refer to other modules:

> However, using the entire library to map a module-name provides too much flexibility and leads to the possibility of name conflicts. Instead the compiler interprets module-names using a directory supplied by the user [Liskov, 1973c, p. 29].

The description units used in this way did not need to contain any implementations. Implementations would be selected in a separate step, at link time. In this way we could support top down program construction, and we could change the implementation of a used module without having to recompile using modules. The library is described in the reference manual [Liskov, 1979c; Liskov, 1984].

The CLU library was never implemented because we never had enough time; it was finally implemented for Argus [Liskov, 1983; Liskov, 1988]. However, our compiler and linker provide an approximation to what we wanted. The compiler can be run in "spec" mode to produce interface specifications of a module or modules and store them in a file. One or more spec files can be supplied when compiling a module and the compiler will use the information in them to do inter-module type checking. Implementations are selected using the linker, which combines (object) files produced by the compiler into a program.

Our insistence on declared interface specifications contrasts with work on type inference, e.g., in ML [Milner, 1990]. I believe specifications are crucial because they make it possible for programmers to work independently; one person can implement an abstraction while others implement programs that use it. Furthermore, the compiler should use the information in the specification since this makes top-down implementation possible. Inference could still be used within the body of a module, however.

4. Evaluation

As mentioned, the main goal of the work on CLU was to contribute to research in programming methodology. We hoped to influence others through the export of ideas rather than by producing a widely used tool. In this section I discuss the success of CLU as a programming tool and a programming language, and also its influence on programming methodology.

CLU has been used in a number of applications including a text editor called TED that is still in use today, a WYSIWYG editor called ETUDE, a browser for database conceptual schemas, a circuit design system, a gate array layout system, and the LP theorem-proving system [Garland, 1990] and other related work in rewriting systems [Anantharaman, 1989]. These projects vary in size; some were large projects involving several programmers and lasting several years. CLU is still being used in the work on LP.

CLU has also been used in teaching; this is probably its main use today both at M.I.T. and elsewhere (e.g., at the Tokyo Institute of Technology where it is "the language" in Information Science department [Kimura, 1992]). It is the basis of a book on programming methodology that I wrote with John Guttag [Liskov, 1986a]. It is used at M.I.T. in our software engineering course and also in our compiler construction course.

In addition, CLU has been used in research. There have been follow-on projects done elsewhere including a CLU-based language developed in Finland [Arkko, 1989], and a parallel version of CLU called CCLU [Bacon, 1988a; Cooper, 1987] developed at Cambridge University in England. CCLU grew out of the Swift project at MIT [Clark, 1985], in which CLU was extended and used as a system programming language. It has been widely used in research at Cambridge [Bacon, 1988b; Craft, 1983]. Also, CLU was the basis of my own later work on Argus [Liskov, 1983; Liskov, 1988], a programming language for distributed systems.

Although CLU has been exported to several hundred sites over the years, it isn't used widely today. In retrospect, it is clear that we made a number of decisions that followed from our view of CLU as a research vehicle but that made it highly unlikely for CLU to succeed in the marketplace. We did not take any steps to promote CLU nor to transfer it to a vendor to be developed into a product. Furthermore, in developing our compiler, we emphasized performance over portability, and the compiler is difficult to port to new machines. (This problem is being corrected now with our new portable compiler.) Finally, we were very pure in our approach to the language; a practical tool might need a number of features we left out (e.g., formatted I/O).

In spite of the fact that it is not widely used, I believe that CLU was successful as a language design. CLU is neat and elegant. It makes it easier to write correct programs. Its users like it (to my surprise, they even like the "t$o" notation because they believe it enhances program correctness and readability). CLU does not contain features that we would like to discard, probably because we were so parsimonious in what we put in. Its features have stood the test of time. It is missing some desirable features including recursive type definitions and a closure mechanism. (Some of these features have been put into Argus.)

CLU has been influential on programming languages both directly and indirectly. Many of the features of CLU were novel; in addition to the support for data abstraction through clusters, there are iterators, the exception mechanism, and the mechanism for polymetric polymorphism. These ideas have had an important impact on programming language design and CLU's novel features have made their way into many modern languages. Among the languages influenced by CLU are Ada, C++, ML, Modula 3, and Trellis/Owl [Schaffert, 1986].

CLU is an object-oriented language in the sense that it focuses attention on the properties of data objects and encourages programs to be developed by considering abstract properties of data. It differs from what are more commonly called object-oriented languages in two ways. The first difference is relatively small: CLU groups operations in types while object-oriented languages group them with objects. The other is more significant: CLU lacks an inheritance mechanism. Object-oriented languages use inheritance for two purposes. Inheritance is used to achieve "subtype polymorphism", which is the ability to design by identifying a generic abstraction and then defining more specific variants of that abstraction as the design progresses (for example, "windows" with "bordered windows" as a subtype). Inheritance is also used to develop code by modifying existing code, and in most object-oriented languages, encapsulation can be violated, because the designer of the subclass can make use of implementation details of the superclass. Of course, this means that if the superclass implementation is changed, all the subclasses will need to be reimplemented. I think this use of inheritance is not desirable in production programs or in programs developed by many people.

I believe that subtype polymorphism is a useful program development idea. If CLU were being designed today, I would probably try to include it. I am doing such a design in my current research on an object-oriented database system called Thor [Liskov, 1992].

The work on CLU, and other related work such as that on Alphard, served to crystallize the idea of a data abstraction and make it precise. As a result, the notion is widely used as an organizing principle in program design and has become a cornerstone of modern programming methodology.

Acknowledgements

I consulted a number of people about historical matters, including Russ Atkinson, Austin Henderson, Jim Horning, Eliot Moss, Greg Nelson, Bob Scheifler, Mary Shaw, Alan Snyder, and Steve Zilles. In addition, several people gave me comments about an earlier draft of this paper, including Mark Day, Dorothy Curtis, John Guttag, Butler Lampson, Eliot Moss, Bob Scheifler, and Alan Snyder, and the referees.

A. People Who Attended the Harvard Meeting

There were about twenty attendees at the Harvard meeting, including: Jack Dennis, Tony Hoare, Joe Stoy, Barbara Liskov, Bill Wulf, Mary Shaw, Steve Zilles, Austin Henderson, James H. Morris, Jim Horning, Carl Hewitt, Doug Ross, Ole-Johan Dahl, John Reynolds, Jim Mitchell, Nico Habermann, and Brian Clark.

B. People Involved in the CLU Effort

CLU originated in joint work between myself and Steve Zilles, with Austin Henderson acting as an interested observer and critic. Most of the work on the CLU design was done by myself, Russ Atkinson, Craig Schaffert, and Alan Snyder, but others also contributed to the design, including Toby Bloom, Deepak Kapur, Eliot Moss, Bob Scheifler, and Steve Zilles. Over the course of the CLU project, the CLU group also included Jack Aiello, Valdis Berzins, Mark Laventhal, and Bob Principato. In addition to members of the CLU group, the CLU meetings in the first two years were attended by Nimal Amersinghe, Jack Dennis, Dave Ellis, Austin Henderson, Paul Kosinski, Joe Stoy, and Eiiti Wada.

The first CLU implementation was done by Russ, Craig, and Alan. Eliot Moss and Bob Scheifler worked on later implementations. Still later, implementation work was done by Paul Johnson, Sharon Perl, and Dorothy Curtis.

C. Project Schedule

From the time the design started in 1973 until we had our production compiler in 1980, I estimate that approximately fourteen person-years were spent on CLU. Until 1978, all of this work was done by myself and students. In June of 1978, Bob Scheifler became full-time technical staff and in March, 1979, Paul Johnson joined the group. By then, the research group was working on the Argus project [Liskov, 1983; Liskov, 1988]. Bob and Paul worked on the CLU implementation, but they also spent part of their time contributing to our work on Argus.

The work on CLU proceeded in several stages:

CLU .5

The first stage was the design and implementation of a preliminary version of CLU called CLU .5. This work started in the fall of 1973. At first language design issues were considered at meetings of a group that included both people interested in CLU and people working on Jack Dennis' dataflow language [Dennis, 1975]. In fact, our initial plan was to use the dataflow work as a basis for the CLU definition [Dennis, 1974], but this plan was dropped sometime in l974. The two groups began to meet separately in Jan. 1974, although members of the data flow group continued to attend CLU meetings. Most of the work between meetings was done by members of the CLU group, especially Russ, Craig, Alan, and myself; Steve and Austin also joined in some of this work.

The goal over the first year was to define a preliminary version of CLU that could be implemented as a proof of concept. Work on the compiler started in summer 1974 and was done by Alan (the parser), Russ (the code generator), and Craig (the type checker). At first the code generator produced Lisp; later, for political reasons, it was changed to produce MDL [Falley, 1977]. (MDL was a dialect of Lisp that contained a richer set of data structures and did some compile-time type checking.) The compiler was initially implemented in Lisp, but was soon rewritten in CLU. Using CLU to implement its own compiler was very helpful to us in evaluating its expressive power. The implementation was done for the PDP-10.

CLU .5 is described in [Liskov, 1974c] and also in the preliminary reference manual, which was published (internally only) in January 1975 [Snyder, 1975]. It included all of current CLU (in some form) except for exception handling and iterators. It includes parameterized types (type definitions that take types as parameters and can be instantiated to produce types), but the mechanism required type checking at runtime. At that point it was unclear to us whether parameterized types really could be type-checked statically.

CLU

At the same time that we were implementing CLU .5, we continued work on the design of CLU. All the features of CLU were designed and integrated into the language by the end of 1976. A paper documenting CLU at this stage appeared in early 1976 [Liskov, 1976] and another one in early 1977 [Liskov, 1977c]. After we felt that we understood every part of CLU, we spent most of 1977 reviewing the design and made lots of small changes, e.g., to the syntax. As we went along, we changed the compiler to match the language. The CLU reference manual was published in July 1978 [Liskov, 1978a].

In 1977, we reimplemented the compiler so that it generated instructions in macro-assembler rather than MDL, leading to both faster runtime execution and faster compilation. (MDL had a very slow compiler and we found the time taken to do double compilation — from CLU to MDL to assembler — very annoying.) Going directly to assembler meant that we had to write our own standalone runtime system, including the garbage collector. In addition, we had to provide our own debugger. In doing the move we designed new implementation techniques for iterators, exception handling, and parameterized modules; these are described in [Liskov, 1978b]. Bob Scheifler did the compiler front end, Russ Atkinson implemented the runtime system (as macros) and the debugger, and Eliot Moss wrote the garbage collector.

Finishing Up

We did a final pass at the language design during 1979. We had quite a bit of user experience by then and we added some features that users had requested, most notably the "resignal" statement (this is part of our exception mechanism), and "own" data. Our last design note appeared in July, 1979. The final version of the reference manual was published in October 1979 [Liskov, 1979c; Liskov, 1984].

The compiler was changed to accept the new features, and also to produce machine code rather than macros; the compiler produced code for Dec System 20. Only at this point did we provide static instantiation of parameterized modules (in the linker, written by Paul Johnson); earlier implementations had used a dynamic approach, in which information about the parameters was passed to the code at runtime. We also finally provided an intermodule type-checking mechanism (see Section 3.11). By 1980 we had a high quality compiler that could be exported to other groups with confidence.

Later we retargeted the compiler for Vaxes and still later for M68000 machines (Sharon Perl did this port). Today we are moving CLU again, but this time we are changing the compiler to generate C so that it will be easy to port; Dorothy Curtis is doing this work.

References

[Aiello, 1974]
Aiello, Jack, *An Investigation of Current Language Support for the Data Requirements of Structured Programming,* Technical Memo MIT/LCS/TM-51, MIT Laboratory for Computer Science, Cambridge, MA, September 1974.

[Anantharaman, 1989]
Anantharaman, S., J. Hsieng, and J. Mzali, SbReve2: A Term Rewriting Laboratory with AC-Unfailing Completion, in *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, 1989. Lecture Notes in Computer Science, 355, Springer-Verlag.

[Arkko, 1989]
Arkko, J., V. Hirvisalo, J. Kuusela, E. Nuutila and M. Tamminen, *XE Reference Manual (XE Version 1.0*), 1989. Dept. of Computer Science, Helsinki University of Technology, Helsinki, Finland.

[Atkinson, 1975]
Atkinson, Russell, *Toward More General Iteration Methods in CLU,* CLU Design Note 54, Programming Methodology Group, MIT Laboratory for Computer Science, Cambridge, MA, September 1975.

[Bacon, 1988a]
Bacon J., and K. Hamilton, Distributed Computing with RPC: The Cambridge Approach, in Barton, M., et al., eds., *Proceedings of IFIPS Conference on Distributed Processing*, North Holland, 1988.

[Bacon, 1988b]
Bacon J., I. Leslie, and R. Needham, Distributed Computing with a Processor Bank, in *Proceedings of Workshop on Distributed Computing*, Berlin: 1988. Also Springer Verlag Lecture Notes in Computer Science, 433, 1989.

[Balzer, 1967]
Balzer, Robert M., Dataless Programming, in *Fall Joint Computer Conference*, 1967.

[Berzins, 1979]
Berzins, Valdis, *Abstract Model Specifications for Data Abstractions,* Technical Report MIT/LCS/TR-221, MIT Laboratory for Computer Science, Cambridge, MA, July 1979.

[Cardelli, 1988]
Cardelli, Luca, A Semantics of Multiple Inheritance, *Information and Computation,* 76, 1988, 138-164.

[Chambers, 1990]
Chambers, Craig, and David Ungar, Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs, in *Proc. of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990.

[Clark, 1985]
Clark, David, The Structuring of Systems Using Upcalls, in *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, Orcas Island, WA: ACM 1985.

[Cooper, 1987]
Cooper, R., Pilgrim: A Debugger for Distributed Systems, in *Proceedings of IEEE 7th ICDCS*, Berlin: 1987.

[Craft, 1983]
Craft, D., Resource Management in a Decentralised System, *Operating Systems Review,* 17:5, June l983, 11-19.

[Dennis, 1974]
Dennis, Jack, and Barbara Liskov, *Semantic Foundations for Structured Programming*, Proposal to

National Science Foundation, 1974.

[Dennis, 1975]
Dennis, Jack, *A First Version of a Data Flow Procedure Language,* Project MAC Technical Memorandum 66, Cambridge, MA: MIT Laboratory for Computer Science, May l975. Also published in *Proceedings of Symposium on Programming*, Institut de Programmation, University of Paris, Paris, France, April l974, 241-271.

[Dijkstra, 1968a]
Dijkstra, Edsger W., Go To Statement Considered Harmful, *Communications of the ACM,* 11:3, March 1968, 147-148.

[Dijkstra, 1968b]
Dijkstra, Edsger W., The Structure of the "THE"-multiprogramming System, *Communications of the ACM,* 11:5, May 1968, 341-346.

[Dijkstra, 1969]
Dijkstra, Edsger W., Notes on Structured Programming, in *Structured Programming*, Academic Press, 1969.

[Earley, 1971]
Earley, Jay, Toward an Understanding of Data Structures, *Communications of the ACM,* 14:10, October 1971, 617-627.

[Falley, 1977]
Falley, Stuart W., and Greg Pfister, *MDL - Primer and Manual*, MIT Laboratory for Computer Science, Cambridge, MA, 1977.

[Garland, 1990]
Garland, Stephen, John Guttag, and James Horning, Debugging Larch Shared Language Specifications, *IEEE Transactions on Software Engineering,* 16:9, September 1990, 1044-1057.

[Goguen, 1975]
Goguen, J. A., J. W. Thatcher, E. G. Wagner, and J. B. Wright, Abstract Data-types as Initial Algebras and Correctness of Data Representations, in *Proceedings of Conference on Computer Graphics, Pattern Recognition and Data Structure*, May 1975.

[Goodenough, 1975]
Goodenough, John, Exception Handling: Issues and a Proposed Notation, *Communications of the ACM,* 18, December 1975, 683-696.

[Guttag, 1975]
Guttag, John, *The Specification and Application to Programming of Abstract Data Types,* Technical Report CSRG-59, Computer Systems Research Group, University of Toronto, Canada, 1975.

[Guttag, 1980]
Guttag, John, Notes on Type Abstraction (Version 2), *IEEE Transactions on Software Engineering,* SE-6:1, January 1980, 13-23.

[Guttag, 1977]
Guttag, John, Abstract Data Types and the Development of Data Structures, *Communications of the ACM,* 20:6, June l977. Also in *Proceedings of Conference on Data: Abstraction, Definition and Structure*, Salt Lake City, UT, March 1976.

[Herlihy, 1982]
Herlihy, Maurice, and Barbara Liskov, A Value Transmission Method for Abstract Data Types, *ACM Transactions on Programming Languages and Systems,* 4:4, October 1982, 527-551.

[Hoare, 1972]
Hoare, C. A. R., Proof of Correctness of Data Representations, *Acta Informatica,* 4, 1972, 271-281.

[Ichbiah, 1973]

Ichbiah, Jean, J. Rissen, and J. Heliard, The Two-Level Approach to Data Definition and Space Management in the LIS System Implementation Language, in *Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting -- Programming Languages-Operating Systems*, Savannah, GA: ACM, April 1973.

[Kimura, 1992]
Kimura, Izumi, *Private Communication*, 1992.

[Lampson, 1974]
Lampson, Butler, James Mitchell, and Edward Satterthwaite, On the Transfer of Control Between Contexts, in *Proceedings of Symposium on Programming*, Institut de Programmation, University of Paris, Paris, France: 1974.

[Landin, 1964]
Landin, Peter, The Mechanical Evaluation of Expressions, *Computer Journal,* 6:4, January l964, 308-320.

[Levin, 1977]
Levin, Roy, *Program Structures for Exceptional Condition Handling*, Ph.D. dissertation, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1977.

[Liskov, 1972a]
Liskov, Barbara, A Design Methodology for Reliable Software Systems, in *Proceedings of Fall Joint Computer Conference 41, Part 1*, IEEE, December 1972. Also published in *Tutorial on Software Design Techniques*, Peter Freeman and A. Wasserman, Eds., IEEE, 1977, 53-61.

[Liskov, 1972b]
Liskov, Barbara, The Design of the Venus Operating System, *Communications of the ACM,* 15:3, March 1972. Also published in *Software Systems Principles: A Survey*, Peter Freeman, SRA Associatues, Inc., Chicago 1975, 542-553.

[Liskov, 1973a]
Liskov, Barbara, Report of Session on Structured Programming, in *Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting -- Programming Languages-Operating Systems*, Savannah, GA: ACM, April 1973.

[Liskov, 1973b]
Liskov, Barbara, Fundamental Studies Group Progress Report, in *Project MAC Progress Report X*, Cambridge, MA: MIT Laboratory for Computer Science 1973.

[Liskov, 1973c]
Liskov, Barbara, and Stephen Zilles, *An Approach to Abstraction,* Computation Structures Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, September 1973.

[Liskov, 1974a]
Liskov, Barbara, and Stephen Zilles, Programming with Abstract Data Types, in *Proceedings of ACM SIGPLAN Conference on Very High Level Languages*, ACM 1974.

[Liskov, 1974b]
Liskov, Barbara, Fundamental Studies Group Progress Report, in *Project MAC Progress Report XI*, Cambridge, MA: MIT Laboratory for Computer Science 1974.

[Liskov, 1974c]
Liskov, Barbara, *A Note on CLU,* Computation Structures Group Memo 112, Laboratory for Computer Science, MIT, Cambridge, MA, November 1974.

[Liskov, 1975a]
Liskov, Barbara, *Multiple Implementation of a Type,* CLU Design Note 53, Cambridge, MA: MIT Laboratory for Computer Science, July 1975.

[Liskov, 1975b]
Liskov, Barbara, Fundamental Studies Group Progress Report, in *Laboratory for Computer Science*

Progress Report XII, Cambridge, MA: MIT Laboratory for Computer Science 1975.

[Liskov, 1976]
Liskov, Barbara, Introduction to CLU, in S. A. Schuman, ed., *New Directions in Algorithmic Languagues 1975*, INRIA, 1976.

[Liskov, 1977a]
Liskov, Barbara, Programming Methodology Group Progress Report, in *Laboratory for Computer Science Progress Report XIV*, Cambridge, MA: MIT Laboratory for Computer Science 1977.

[Liskov, 1977b]
Liskov, Barbara, and Alan Snyder, *Structured Exception Handling,* Computation Structures Group Memo 155, MIT Laboratory for Computer Science, Cambridge, MA, December 1977.

[Liskov, 1977c]
Liskov, Barbara, Alan Snyder, Russell Atkinson, and J. Craig Schaffert, Abstraction Mechanisms in CLU, *Communications of the ACM,* 20:8, August 1977, 564-576. Also published as Computation Structures Group Memo 144-1, MIT Laboratory for Computer Science, Cambridge, MA, January 1977.

[Liskov, 1978a]
Liskov, Barbara, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder, *CLU Reference Manual,* Computation Structures Group Memo 161, MIT Laboratory for Computer Science, Cambridge, MA, July 1978.

[Liskov, 1978b]
Liskov, Barbara, Russell Atkinson, and Robert Scheifler, Aspects of Implementing CLU, in *Proceedings of the Annual Conference*, ACM 1978.

[Liskov, 1979a]
Liskov, Barbara, *Modular Program Construction Using Abstractions,* Computation Structures Group Memo 184, MIT Laboratory for Computer Science, Cambridge, MA, September 1979.

[Liskov, 1979b]
Liskov, Barbara, and Alan Snyder, Exception Handling in CLU, *IEEE Transactions on Software Engineering,* SE-5:6, November 1979, 546-558.

[Liskov, 1979c]
Liskov, Barbara, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder, *CLU Reference Manual,* Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, Cambridge, MA, October 1979.

[Liskov, 1983]
Liskov, Barbara, and Robert Scheifler, Guardians and Actions: Linguistic Support for Robust, Distributed Programs, *ACM Transactions on Programming Languages and Systems,* 5:3, July 1983, 381-404.

[Liskov, 1984]
Liskov, Barbara, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder, *CLU Reference Manual*, Springer-Verlag, 1984. Also published as Lecture Notes in Computer Science 114, G. Goos and J. Hartmanis, Eds., Springer-Verlag, 1981.

[Liskov, 1986a]
Liskov, Barbara, and John Guttag, *Abstraction and Specification in Program Development*, MIT Press and McGraw Hill, 1986.

[Liskov, 1986b]
Liskov, Barbara, and John Guttag, *Chapter 4, Data Abstraction*, 1986. In [Liskov 1986a].

[Liskov, 1988]
Liskov, Barbara, Distributed Programming in Argus, *Communications of the ACM,* 31:3, March 1988, 300-312.

[Liskov, 1992]
Liskov, Barbara, *Preliminary Design of the Thor Object-Oriented Database System,* Programming Methodology Group Memo 74, MIT Laboratory for Computer Science, Cambridge, MA, March 1992.

[McKeag, 1973]
McKeag, R. M., Programming Languages for Operating Systems, in *Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting -- Programming Languages-Operating Systems*, Savannah, GA: ACM, April 1973.

[Milner, 1990]
Milner, Robin, M. Tofte, and R. Harper, *The Definition of Standard ML*, Cambridge, MA: MIT Press, 1990.

[Mitchell, 1978]
Mitchell, James G., W. Maybury, and R. Sweet, *Mesa Language Manual,* Technical Report CSL-78-1, Xerox Research Center, Palo Alto, CA, February 1978.

[Morris, 1973a]
Morris, James H., Jr., Protection in Programming Languages, *Communications of the ACM,* 16:1, January 1973, 15-21.

[Morris, 1973b]
Morris, James H., Jr., Types are Not Sets, in *Proceedings of the Symposium on Principles of Programming Languages*, ACM 1973.

[Moss, 1978]
Moss, J. Eliot, *Abstract Data Types in Stack Based Languages,* Technical Report MIT/LCS/TR-190, Cambridge, MA: MIT Laboratory for Computer Science, February 1978.

[Palme, 1973]
Palme, Jacob, *Protected Program Modules in Simula 67,* FOAP Report C8372-M3 (E5), Stockholm, Sweden: Research Institute of National Defence, Division of Research Planning and Operations Research, July l973.

[Parnas, 1971]
Parnas, David, Information Distribution Aspects of Design Methodology, in *Proceedings of IFIP Congress*, North Holland Publishing Co., 1971.

[Parnas, 1972a]
Parnas, David, On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM,* 15:12, December 1972, 1053-1058.

[Parnas, 1972b]
Parnas, David, A Technique for the Specification of Software Modules with Examples, *Communications of the ACM,* 15, May 1972, 330-336.

[PMG, 1979a]
Programming Methodology Group, *CLU Design Notes*, MIT Laboratory for Computer Science, Cambridge, MA, 1973-1979.

[PMG, 1979b]
Programming Methodology Group, *CLU Design Meeting Minutes*, MIT Laboratory for Computer Science, Cambridge, MA, 1974-1979.

[Randell, 1969]
Randell, Brian, Towards a Methodology of Computer Systems Design, in P. Naur and B. Randell, ed., *Software Engineering*, NATO Science Committee, 1969.

[Ross, 1970]
Ross, Douglas T., Uniform Referents: An Essential Property for a Software Engineering Language, in J. T. Tou, ed., *Software Engineering*, Academic Press, 1970.

[Schaffert, 1978]
Schaffert, J. Craig, *A Formal Definition of CLU,* Technical Report MIT/LCS/TR-193, MIT Laboratory for Computer Science, Cambridge, MA, January 1978.

[Schaffert, 1986]
Schaffert, Craig, T. Cooper, B. Bullis, M. Kilian and C. Wilpolt, An Introduction to Trellis/Owl, in *Proceedings of ACM Conference on Object Oriented Systems, Languages and Applications*, Portland, OR: September 1986.

[Scheifler, 1976]
Scheifler, Robert, *An Analysis of Inline Substitution for a Structured Programming Language*, S.B. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1976.

[Scheifler, 1977]
Scheifler, Robert, An Analysis of Inline Substitution for a Structured Programming Language, *Commuications of the ACM,* 20:9, September 1977.

[Scheifler, 1978]
Scheifler, Robert, *A Denotational Semantics of CLU,* Technical Report MIT/LCS/TR-201, MIT Laboratory for Computer Science, Cambridge, MA, June 1978.

[Shaw, 1976]
Shaw, Mary, William Wulf, and Ralph London, Carnegie Mellon University and USC Information Sciences Institute Technical Reports, *Abstraction and Verification in Alphard: Iteration and Generators*, August 1976.

[Shaw, 1977]
Shaw, Mary, William Wulf, and Ralph London, Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators, *Communications of the ACM,* 20:8, August 1977.

[Shaw, 1981]
Shaw, Mary, ed., *ALPHARD: Form and Content*, Springer-Verlag, 1981.

[Snyder, 1975]
Snyder, Alan and Russell Atkinson, *Preliminary CLU Reference Manual,* CLU Design Note 39, Programming Methodology Group, MIT Laboratory for Computer Science, Cambridge, MA, January 1975.

[Spitzen, 1975]
Spitzen, Jay, and Ben Wegbreit, The Verification and Synthesis of Data Structures, *Acta Informatica,* 4, 1975, 127-144.

[Wegbreit, 1972]
Wegbreit, Ben, D. Brosgol, G. Holloway, Charles Prenner, and Jay Spitzen, *ECL Programmer's Manual*, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1972.

[Wegbreit, 1973]
Wegbreit, Ben, *The Treatment of Data Types in EL1*, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1973.

[Wirth, 1971]
Wirth, Niklaus, Program Development by Stepwise Refinement, *Communications of the ACM,* 14:4, April l971, 221-227.

[Wulf, 1973]
Wulf, William, and Mary Shaw, Global Variables Considered Harmful, *SIGPLAN Notices,* 8:2, February 1973, 28-34.

[Wulf, 1976]
Wulf, William, Ralph London, and Mary Shaw, An Introduction to the Construction and Verification of

Alphard Programs, *IEEE Transactions on Software Engineering,* SE-2:4, December 1976, 253-265. Presented at Second International Conference on Software Engineering, October 1976.

[Zilles, 1973]
Zilles, Stephen, Procedural Encapsulation: A Linguistic Protection Technique, in *Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting -- Programming Languages-Operating Systems*, Savannah, GA: ACM, April 1973.

[Zilles, 1974a]
Zilles, Stephen, Computation Structures Group Progress Report, in *Project MAC Progress Report XI*, Cambridge, MA: MIT Laboratory for Computer Science 1974.

[Zilles, 1974b]
Zilles, Stephen, *Working Notes on Error Handling,* CLU Design Note 6, Cambridge, MA: MIT Laboratory for Computer Science, January 1974.

[Zilles, 1975]
Zilles, Stephen, *Algebraic Specification of Data Types,* Computation Structures Group Memo 119, Cambridge, MA: MIT Laboratory for Computer Science, March l975.

Table of Contents

List of Figures