AFOSR-TR- 78-1251

LEVEL II

CMU-CS-78-120

(2)

The Evolution of Abstraction in Programming Languages

Loretta Rose Guarino
Computer Science Department
Carnegie-Mellon University

22 May 1978

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR- 78-1251	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): THE EVOLUTION OF ABSTRACTION IN PROGRAMMING LANGUAGES	5. TYPE OF REPORT & PERIOD COVERED: Interim rept.
6. PERFORMING ORG. REPORT NUMBER: CMU-CS-78-120	
7. AUTHOR(s): Loretta Rose Guarino	8. CONTRACT OR GRANT NUMBER(s): F44620-73-C-0074 ARPA Order 2466
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61101E, A02466/7
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, Virginia 22209	12. REPORT DATE: May 22, 1978
	13. NUMBER OF PAGES: 42
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

11 22 May 78 / 16 2466 / 17 07

Approved for public release; distribution unlimited.

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

12 44p.

18. SUPPLEMENTARY NOTES

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

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

As our understanding of the role of abstraction in programming has improved, programming languages have evolved in their use and support of abstraction. This paper defines abstraction and discusses how the use of abstraction in programming languages assists the programmer. It traces in depth the development of support for the abstraction of objects and for the abstraction of control constructs in programming languages.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 089

CMU-CS-78-120

(2)

The Evolution of Abstraction in Programming Languages

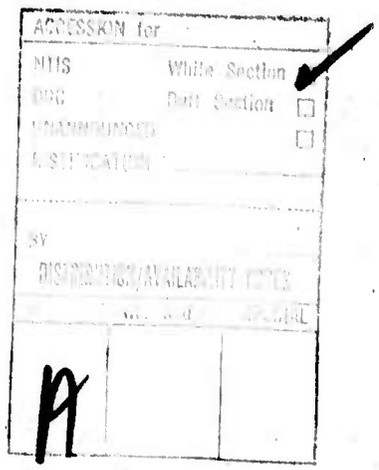
Loretta Rose Guarino
Computer Science Department
Carnegie-Mellon University

22 May 1978

This work was supported in part by a National Science Foundation Fellowship, by a Xerox Corporation Fellowship, and by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074.

Abstract

As our understanding of the role of abstraction in programming has improved, programming languages have evolved in their use and support of abstraction. This paper defines abstraction and discusses how the use of abstraction in programming languages assists the programmer. It traces in depth the development of support for the abstraction of objects and for the abstraction of control constructs in programming languages.



1. Introduction

Abstraction is the distillation of the essential qualities of a collection of items from its individual members. The use of abstraction has been recognized in recent years as a powerful tool in programming -- Dijkstra [10] recognizes abstraction, sequencing, conditionality and iteration as the basic tools of program design, and various languages have been designed to support user creation of abstract objects [33, 47]. The evolution of programming languages reflects the growth of the understanding that abstraction plays a crucial role in programming. This recognition spurred the development of language mechanisms that exploit the power of abstraction.

The first programming languages were binary machine codes. A program was a sequence of bits comprehensible only to someone intimately familiar with the machine hardware. A programmer dealt with specific memory addresses and operation codes.

In the early 1950's, assembly languages introduced an important tool to assist the programmer: naming. A programmer used mnemonic operation codes rather than the binary encoding of the operation; he assigned a name to a jump location or area of data storage, using the name instead of the memory address throughout the program. Naming provided valuable bookkeeping assistance as well as a means for the programmer to express his intentions by the mnemonic choice of names.

The late 1950's and early 1960's saw an important step in the evolution of programming languages: abstraction away from the machine. Programming languages provided a virtual machine more amenable than the bare hardware of the real machine to the problem at hand. FORTRAN [1] presented an environment for scientific calculations in a style similar to scientific notation. It introduced primitive control abstractions such as DO loops and basic data abstractions such as real and integer variables. ALGOL 60 [36] continued this abstraction away from the machine, providing improved control constructs such as block structuring, and data abstractions through typed variables.

The next landmark in language development was to recognize of the importance of abstraction and to support user definitions of their own abstractions. Dijkstra [10] and others pointed out the importance of the use of abstraction in program design. Languages such as ALGOL 68 [3] and SIMULA 67 [7] provided mechanisms that could be used for the user definition of data abstractions. Users defined the representation of data object types.

PASCAL [41] was introduced a few years later to try to retain the advantages of data types in a simpler language.

In the early to middle 1970's, languages were introduced that permitted the user to enforce the correct usage of abstract data types. CLU [33] and ALPHARD [47] provide language features in which a user defines the representation of abstract objects and the operations applicable to the objects, and that protects the objects from misuse by other parts of the program.

The late 1970's find abstraction mechanisms used to address programming problems other than program design. EUCLID [27] was designed for the writing of verifiable programs. MESA [15] uses abstraction mechanisms to aid in the construction of large programmed systems. MODULA [44] and CONCURRENT PASCAL [6] address problems of operating system construction and synchronization.

In the following sections, we define abstraction and discuss how the use of abstraction in programming languages assists the programmer. We then trace the development of support for the abstraction of objects and the abstraction of control constructs in programming languages in light of the evolution of language described above.

2. Abstraction

Much of the success and widespread acceptance of higher-level languages has been attributed to their support of abstraction [19]. In the following sections, we define abstraction and discuss its effect on the act of programming. Explicit recognition of the importance of abstraction has occurred only within the last decade. Hence, although we shall review uses of abstraction over the entire development of programming languages, most references will be to relatively recent sources.

2.1. What we mean by abstraction

Abstraction is the generalization from a collection of objects that all of the items in the collection share some properties that are important for a given purpose. Hoare* states:

> "Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate on these similarities and to ignore for the time being the differences."

Given a collection consisting of a book, paper and a pencil, we might, for example, form the abstraction "burnable item" if we were interested in starting a fire, or the abstraction "school supplies" if we were preparing for school. The abstraction drawn depends upon the properties that we judge to be important, and conversely, we can deduce the intended use of a collection from the abstraction used to describe it.

Wegner defines abstraction [40]:

> "An abstraction of a object is a characterization of the object by a subset of its attributes... If the attribute subset captures the 'essential' attributes of the object, then the user need not be concerned with the object itself but only with the abstract attributes."

We form abstractions because of certain regularities or similarities among the particular instances of the abstraction. The abstraction is useful because we can concentrate on these important features and ignore the dissimilarities between the instances as being incidental. Abstraction permits the expression of relevant details and the suppression of irrelevant ones [32].

---

*References to Hoare are from [19].

2.2. Why abstraction is useful to the programmer

Whether they realized it or not, programmers have always used abstraction as part of the problem-solving process. Data abstraction is used in viewing a sequence of bits as an integer, real number or character [22]. A subroutine call can be thought of abstractly as a machine instruction [23]. The process of programming can be viewed as building up more useful objects from the objects already at hand, or as transforming one machine into another, more suitable machine [11, 31].

Since programming, particularly the programming of large systems, is an extremely complicated task, the programmer uses abstraction as an organizational tool. He uses abstraction mechanisms to decompose problems into subparts; at any level of decomposition, he need only use the important characteristics of the lower level abstractions, and can focus on providing the proper characteristics to higher levels [32]. When using an abstraction, the programmer can understand what the abstraction represents without worrying about how its does so; when implementing an abstraction the programmer can understand how the abstraction is to be represented without worrying about why is it to be created [23].

Wegner's definition of abstraction notes that, if the attribute subset defining an abstraction is substantially simpler than its instance, then the use of the abstraction in place of an instance simplifies the problem addressed by the user. Wirth finds procedures a means of partitioning and structuring programs into logically coherent, closed components in a way that is essential to the understanding of the program [43]. Liskov indicates that programming problems can be solved by means of step-wise abstraction, implementing appropriate data abstractions that are in turn developed by means of subsidiary abstractions [33].

Stepwise refinement [43] employs abstraction in a systematic way in solving problems. In the early stages, the programmer pays attention to global problems rather than to details. As the design progresses, the problem is split into subproblems, and gradually more consideration is given to details of the subproblem specifications and the characteristics of the tools available. Hence the programmer concentrates only on the details relevant to a given stage of the solution.

2.3. Abstraction introduced into programming languages

Explicit abstraction mechanisms in programming languages were introduced to support and encourage the programmer's use of abstraction in program design. A side effect was to improve the ability of the languages to represent the programmer's mental model of a program. A programming language provides the human-readable representation of programs. Hoare relates the importance of representation:

> "The primary use for representations is to convey information about important aspects of the real world to others, and to record this information in written form, partly as an aid to memory and partly to pass it on to future generations."

Programming languages that explicitly display abstraction permit the programmer to indicate what he considers to be the important aspects of a program, aiding its readability and comprehensibility. Such support aids a person in retracing the design process and understanding the programmer's goals and intentions. Ideally, a programming language enables a programming abstraction to be represented easily and naturally, and enables the original programming abstraction to be easily reconstructed from its representation [25]. An appropriate use of abstraction indicates a mentally manageable decomposition of the program. As Hoare indicates, such an improved representation assists in program development, reminding the programmer of previous design decisions, as well as assisting program maintenance.

Hoare suggests several other advantages accruing from the support of explicit abstraction by programming languages. The use of abstraction contributes to the machine independence of programs, since only the implementation of the abstractions, not the abstractions themselves, need be changed between machines [22]. Abstraction mechanisms also help reduce the scope of programming error, both by encapsulation of data [37] and by the regular decomposition of control flow.

2.4. Design advantages accruing from abstraction

The most important effect of the presence of abstractions in programming languages is their influence on the direction of program design. Hoare suggests that programming languages can and do contribute to the program design process by their support of abstraction:

> "The role of abstraction in the design and development of computer programs may be reinforced by the use of a suitable high-level programming language. Indeed, the benefits of using a high-level language instead of machine code may be largely due to the incorporation of successful abstractions, particularly for data."

The presence of certain abstractions shapes the way we approach problems. As an example of the way in which different languages affect our thinking about a problem by the abstractions they provide, we consider the task of calculating the area of a region [38, 39]. We are given a two-dimensional space of squares, each square painted either black or white. The black squares form a connected region, dividing the set of white cells into isolated regions. We wish to calculate the area of one such white region, given the location of a cell known to be within the region.

A language supporting recursion, as does ALGOL, lends itself to a recursive solution. A starting cell is counted and painted black, and we recur for each of its four immediate neighbors. A cell is counted and its neighbors inspected only if it is white. Note that a cell can be black either because it was not part of the original region, or because it has already been counted and painted black.

We might be led to an "edge-following" solution in a language like FORTRAN, which most strongly supports iteration. We calculate the area of the region by tracing its boundary. The area of the region will be the sum of the lengths of each vertical column of cells. We can calculate these lengths by our knowledge of the boundary location.

Using APL, a language providing simultaneous operations on arrays, we might solve this problem by considering a kernel of the space known to be within the region. At each step, we "grow" the kernel to include all white cells adjacent to its boundary, continuing to expand until the kernel fills the entire region. The area is the number of cells grown. We can "grow" an area in a single step quickly and easily because APL permits us to shift and combine entire arrays with single operations.

Since the abstractions provided can have such a strong influence upon language design, the introduction of user-defined abstractions provided design flexibility while retaining the other advantages of abstraction. We can view every large programming project as involving the design, use and implementation of a special-purpose programming language, with its own data concepts and primitive operations, specifically oriented to that project [20]. Abstraction mechanisms permit the programmer to extend his language, either for a particular program or for some application area [23]. The programmer can tailor a language toward his design

needs, instead of letting his design be driven by the abstractions provided by a language. He retains and enhances the other advantages of abstraction, organization, improved representation, and portability.

3. The Abstraction of Objects

The development of the abstraction of objects in programming languages has closely mirrored the language evolution described in the introduction. Each advance in language evolution has been accompanied by a major improvement in our understanding and support of object abstraction. The ability to name objects was a major improvement of assembly language over machine language. Variable declaration and types accompanied abstraction away from the machine. With the explicit recognition of the importance of abstraction in the sixties came facilities for users to define their own object types. The advances of the seventies have guided the development of facilities for user definition of abstract objects.

3.1. Basic support: names

The earliest tool provided to the programmer to express abstractions, introduced in assembly languages, was the ability to name things. Naming was introduced to reduce the amount of bookkeeping involved in programming. It permits the programmer to abstract away from machine addresses, so he can concern himself with the use of objects rather than their locations. Naming shields the programmer from many of the bookkeeping details of the program because the translator that maps the names into memory addresses can easily keep track of these details. If a program includes the statement

```
JUMP    100
```

to transfer control to the statment at location 100, then any modifications that cause instructions to be added before location 100 will require that the JUMP statement be changed. If the programmer can instead write

```
JUMP    TERM
```

where TERM is a name attached to the desired instruction, then the translator can make the adjustments when new code is added. Since the translator deals with the side effects of moving, adding, or deleting objects, the programmer can turn his attention to the use of the objects.

In addition to this bookkeeping ability, the programmer gained the ability to use names mnemonically. The programmer could name objects to reflect the manner in which he intended to use them. This alone makes programs immensely more understandable.

Languages do not enforce any consistency between use and intention -- mnemonic names are a means of communicating between people, not between person and computer -- but the naming ability enabled the programmer to reflect his intended usage, for his benefit and for the benefit of others reading his program.

The ability to give names to things was first introduced in assembly languages, although names were often restricted in length or format. The restrictions on name length hampered the use of mnemonic names, and acronyms abounded. But the restrictions on name length and available characters were slowly raised, to the point that some languages support arbitrarily long names, multiple fonts, upper and lower cases, and spacing conventions within names. Multi-word names are often realized by judicious use of upper and lower case, e.g., CharactersPerLine, or by use of spacing characters, e.g., CHARACTERS.PER.LINE or CHARACTERS_PER_LINE.

3.2. Fixed abstractions

3.2.1. Language designers provide application oriented data abstractions

The next step in the support of abstraction after providing a naming ability was to supply the user with the built-in or implicit abstract objects that the language designer felt were useful. Different languages provided different implicit abstractions, and hence were more appropriate for some applications than others.

FORTRAN was designed to assist in scientific computations. It introduced a form of common scientific notation for arithmetic expressions. Later versions supported extended-precision and complex arithmetic for use in scientific calculations. FORTRAN's arrays can be used as a matrix abstraction, although FORTRAN provides no primitive operation whose scope is an array or matrix. Hence, the following FORTRAN program fragment performs a matrix addition operation on the two-dimensional arrays being used to represent matrices:*

---

*Example programs are presented to show typical language syntax and use of abstraction. No attempt is made to define or describe languages in detail.

```
      DIMENSION A(99,99),B(99,99),C(99,99)
      ...
      DO 10 I = 1,99
        DO 20 J = 1,99
          C(I,J) = A(I,J) + B(I,J)
20      CONTINUE
10    CONTINUE
```

COBOL [2] presents a rich set of abstractions useful for commercial data processing and unit-record oriented programming. The set is intellectually manageable because of the structure of the abstractions provided; each data type is composed of a series of attributes, with a choice of several values for each attribute. The language contains a rich, but incomplete, set of editing rules that convert data from one type to another. For instance, COBOL permits conversion from integer to string, but not from string to integer. COBOL programmers learn to take advantage of the built-in coercions to accomplish much of their computation:

```
DATA DIVISION.
FILE SECTION.
FD  LEDGER-FILE DATA RECORD IS OUTPUT-LEDGER-RECORD.
01  OUTPUT-LEDGER-RECORD.
    02 OUTPUT-SOCIAL-SEC.
        03 PART1            SIZE 3 USAGE DISPLAY.
        03 FILLER           SIZE 1 VALUE "-".
        03 PART2            SIZE 2 USAGE DISPLAY.
        03 FILLER           SIZE 1 VALUE "-".
        03 PART3            SIZE 4 USAGE DISPLAY.
    02 OUTPUT-SALARY        PICTURE $$$,$$9.99.

WORKING STORAGE SECTION.

01  INTERNAL-SOCIAL-SEC.
    02 PART1                USAGE COMPUTATIONAL SIZE 3.
    02 PART2                USAGE COMPUTATIONAL SIZE 2.
    02 PART3                USAGE COMPUTATIONAL SIZE 4.

77  WORKER-SALARY           USAGE COMPUTATIONAL PICTURE 9999V99.
```

```
. . . .
PROCEDURE DIVISION.
. . . .
PRINT-LEDGER-RECORD.
    MOVE CORRESPONDING INTERNAL-SOCIAL-SEC TO OUTPUT-SOCIAL-SEC.
    MOVE WORKER-SALARY TO OUTPUT-SALARY.
    WRITE LEDGER-RECORD.
    GOTO NEXT-INPUT.
```

This use of data abstractions and coercions is useful and manageable because the data types are built-in and not extendable and are built up in a regular fashion, so the programmer can master their use.

SNOBOL [17] is intended for text processing, so it provides the abstractions of character strings and patterns. The SNOBOL string implementation is particularly flexible. A string can be arbitrarily long, and text can be added or removed from anywhere within a string. Powerful pattern-matching operations permit strings to be easily and intricately manipulated. For example, we can create a SNOBOL pattern that will take a subject string STR and a pattern PAT and find the longest substring of STR that PAT matches:

```
MAXPAT  =  (*PAT $ TRY *GT(SIZE(TRY)),SIZE(BIG))) $ BIG FAIL
```

BIG must be initialized to the null string before using this pattern, and will contain the maximal substring after matching, as in the following program fragment:

```
PAT  =  SPAN('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
STR  =  'THIS IS A SAMPLE TEXT SENTENCE'
BIG  =
STR MAXPAT
OUTPUT  =  'LARGEST WORD IS '  BIG
```

which produces the output

```
LARGEST WORD IS SENTENCE
```

LISP [34] is both a mathematical formalism and a programming language for describing computations with symbolic expressions. It was designed for use in artificial intelligence applications, and it incorporated many of the abstractions used in formal logic. LISP relates symbols by means of lists, which are the basic abstraction provided by the language. The recursive character of LISP's control constructs melds neatly with the list abstraction. A typical use of recursion and the list abstraction in LISP is demonstrated by the following

definition of the function REVERSE, which reverses the order of the elements of its list parameter.

```
(DE REVERSE (L) (REVERSE1 L NIL))
(DE REVERSE1 (L M)
      (COND ((ATOM L) M)
            (T (REVERSE1 (CDR L) (CONS (CAR L) M)))))
```

The basic abstraction provided by APL [24] is the array. Operators are provided to transform and manipulate arrays of arbitrary dimension and to reshape and redimension arrays. Character strings are treated as one-dimensional arrays whose elements are characters. The following APL statement will remove all extra blanks from an array of characters S:

```
S ← (C .OR. 1 .RV. C ← S .NE. ' ')/S
```

As we mentioned in the area-finding problem, the basic abstractions provided by a language affect the types of programs written easily in that language. The languages described above present widely differing programming environments because of the basic abstractions they support.

3.2.2. Built-in abstractions reflect underlying hardware

Many of the initial decisions about the particular data abstractions to be included in a programming language were strongly influenced by the capabilities of the machine for which they were initially intended. The form of FORTRAN, for example, was oriented to the IBM 701 computer [4, 43]. Language designers wanted to be able to use the underlying hardware as efficiently as possible. The features and peculiarities of the computer strongly affected the abstractions provided, as well as the implementations of those abstractions.

Early languages were designed around models of implementations derived from the designer's intended use of the hardware [40]. FORTRAN's model of implementation required that storage requirements be known at compile time so all addresses could be assigned then. This static storage allocation engendered FORTRAN's lack of recursion. ALGOL 60's implementation model included dynamic storage allocation by means of a run-time stack, which permits recursive declarations. ALGOL supports arrays whose sizes are not known until execution time, since it can allocate storage dynamically on its stack. However, ALGOL cannot

support the dynamic creation of variables with arbitrarily long, overlapping lifetimes, as is possible in LISP.

Most languages include the abstractions integer and real, which are implemented by means of the machine's fixed- and floating-point numbers. These abstractions differ from the mathematical notion of integer and real numbers, of course, because the fixed-point numbers are restricted to a finite subrange of the integers and floating point numbers can represent only a sparse sampling of the real numbers. For most applications, this implementation of the abstractions of integer and real numbers is acceptable. Extended-precision packages have been developed to provide implementations that correspond more closely to the abstractions. The need to know when and how badly these implementations differ from the abstractions in use has stimulated the development of the discipline of numerical analysis.

The mapping between a higher-level abstraction and the hardware implementing it gave rise to the use of the names CAR and CDR in LISP. CAR refers to the first element of a list and CDR to the rest of the list. Clearly, these names are not at all indicative of their functions. The first LISP implementation was on an IBM 704, a 36 bit computer whose instruction word was divided into four subfields, with a special instruction to access the contents of each field. The "address" and "decrement" fields, each 15 bits, were used to hold the two pointers that made up a list cell, and the LISP operations CAR and CDR were named for the 704 operations that implemented them: "Contents of Address field to Register" and "Contents of Decrement field to Register".

The restrictions on FORTRAN subscript expressions were designed to make the best use of early indexing hardware [4]:

```
A(10*I-3)
```

would be a permissible expression in FORTRAN, but

```
A(I*J)
```

would not. A subscript expression could be of the form

{<integer constant>*} <integer scalar variable> {+<integer constant>}

where all constants were limited to 15 bits. This was the least restrictive syntax that would allow strength reduction in loops and would allow the Variable Length Multiply Instruction to be used to compute non-loop subscripts. Strength reduction, in which multiplications in the calculation of indices are replaced by additions within loop, was necessary because the index

register was separate from the arithmetic register and could perform no arithmetic operations except addition. FORTRAN's array performance would have been degraded severely if it had been necessary to use the arithmetic register to perform index calculations within loops.

The FORTRAN arithmetic IF statement reflects a peculiarity of the IBM 704 instruction set. The only comparison instruction on the 704 was CAS (Compare Accumulator to Storage), which would skip no commands if the accumulator was greater than the storage value, skip one instruction if the accumulator was equal to storage, and skip 2 instructions if the accumulator was less than storage. The FORTRAN arithmetic IF would branch to different locations if the value of its expression was greater than, equal to or less than zero. It was designed to exploit the presence of the CAS instruction.

3.2.3. Limitations of built-in abstractions

The languages discussed provide abstractions that are quite useful for their intended purposes. As testified to by their continued heavy usage, these languages present a major increase in convenience and clarity to the programmer by their abstraction away from the underlying machine. However, no language is appropriate for all applications. The programmer is limited to the abstractions that a language provides. Attempting to manipulate text in FORTRAN is frequently even more difficult than in assembly language. Furthermore, it is difficult to join together subprograms from different languages, so one can not simply split up a task into the languages most appropriate for each part.

3.2.4. Proliferation of built-in abstractions

PL/1 [28] takes the use of built-in abstraction to its extreme. It attempted to become the ultimate general-purpose language, suitable for commercial, scientific and real-time applications, by collecting many of the abstractions used by other languages. Unfortunately, this primarily resulted in overwhelming the user. PL/1 lacks a systematic structure with a unifying underlying conception [43] and the user faces a wide range of abstractions from which to select. Furthermore, complex coercion and default rules are built into the language to define the interactions between the abstractions. They are numerous, often non-intuitive, the programmer has no control over them, and he seldom receives warning when they are applied. As an example, in the following program

```
DECLARE B BIT(1);
B=1;
IF B=1 THEN GOTO Y;
       ELSE GOTO X;
```

execution transfers to label X because the fixed-decimal constant 1 is converted to a bit data type in the assignment statement and to a binary data type in the comparison [35].

Wegner [40] claims that programming in PL/1 is relatively easy once the language is mastered, but the complexity of the language makes this mastery difficult and makes verifiability and readability of programs a problem. Because of the complexity of the abstraction interactions, the programmer loses the usual advantages of abstraction -- intellectual manageability and minimization of arbitrary interactions.

3.3. Languages support programmer creation of new abstractions

The futility of providing a language with too many implicit abstractions caused language designers to step back and discover that a recursive application of the principle of abstraction was in order. Programming languages provide specific abstractions as tools from which to build. Let one of the objects provided be the concept of abstraction itself: provide the programmer with the basic essentials from which to build and let him create his own more complex or specialized abstractions as needed.

3.3.1. Type checking and the ability to create new types

By defining his own data types, the programmer can specify the representation of his abstract objects. SIMULA 67, with its classes, and ALGOL 68, with its modes, were among the first languages to permit the programmer to localize the definition of an abstract object's representation. The programmer gives the abstract object definition a unique type name, which he then uses to declare variables in his program. In this way, changes to the representation need only be made in the type definition for the programmer to be assured that all instances of a type of abstract object possess the same representation.

One of the important bookkeeping aids provided by a programming language is type checking, that is, the consistency checking of operands [20]. Operations may only be performed upon operands of the correct or consistent type; we cannot add apples and

oranges. Languages assist the programmer in the use of new abstractions by performing type checking on programmer-defined as well as implicit types of operands.

What we shall call weak type checking ensures that operands have identical underlying representations. A complex number and a point on the plane can both be represented by a pair of real numbers. If we declare two types,

```
TYPE Complex = RECORD [RealPart, ImagPart: Real];
TYPE Point   = RECORD [XCoord, YCoord: Real];
```

languages with weak type checking will not distinguish between variables of these types. We can test whether a variable of type *Complex* is equal to a variable of type *Point*, for instance. In effect, the language only checks whether the structure of an operand permits such an operation. Weak type checking is useful in writing operations such as an output formatter, for which the underlying representation is the important characteristic of the operand. It does not assist the programmer in enforcing the distinction between separate types with identical representations. Weak type checking is used by SIMULA 67, ALGOL 68, and PASCAL, the early languages to introduce user-defined data types.

MESA, ALPHARD and CLU enforce strong or strict type checking [15, 32], which distinguishes between usages of separate types with identical representations. Languages with strong type checking ignore the underlying representations of objects. Objects of different types are conceptually different and should be treated as separate entities. In such languages, objects of types *Complex* and *Point* cannot be confused or interchanged. Hence it would be impossible to accidentally square a *Point* variable or compare a *Complex* variable with a *Point* variable.

To be efficient and to reduce the proliferation of procedures that are identical except for the types of their parameters, languages such as SIMULA, EUCLID, CLU and ALPHARD support forms of parameterized types. Instead of insisting that an operand be of a certain type, operations can specify that an operand type must possess certain properties. We can use such a facility in a strongly typed language to gain the flexibility of weakly typed languages while retaining type protection. For instance, we can write our output formatter that should accept both *Complex* and *Point* variables by requiring that the operand type possess operations *FirstRealPart* and *LastRealPart* that return the two real parts of the operand. We can use parameterized types to avoid having to create different stack types for every type of object we wish to stack [47]. Instead of creating types *StackOfInteger* and *StackofReal*,

we can create a parameterized type *Stack* that can be used with any type.

3.3.2. Trend toward simplicity

SIMULA 67 and ALGOL 68, the early languages that provided facilities for user-defined types, are large, complex languages. Attempts to achieve greater richness by synthesis of existing features, a la PL/1, and by generalization, a la ALGOL 68 led to excessively elaborate languages. Flexibility and power of expression in programming languages are accompanied by greater complexity [40]. Programming methodologies have been developed that argue for the systematic use of a small set of basic tools in programming development [10, 20]. The desire for correct, reliable programs has not only stimulated program design methodology, but has also increased interest in program verification, both manual and automatic. These developments encouraged the design of simpler languages that would be manageable by the programmer and tractable for verification, even at the price of restricting flexibility and power of expression.

The language PASCAL was designed with the stated goal of providing "a notation in which the fundamental concepts and structures of programming are expressible in a systematic, precise, and appropriate way". It grew out of the desire to retain the advantages of abstraction as developed in SIMULA 67 and ALGOL 68 within a simpler language, based on the spirit of the considerations outline above. It attempts to provide rich data and control abstraction facilities without sacrificing efficiency of implementation. PASCAL provides data abstractions such as sets, enumerated types, and the discriminated union of types by means of variant records. To simplify the implementation of PASCAL, however, arrays must be of a fixed size; the programmer cannot determine his array size at runtime as he could in ALGOL 60. An axiomatic language definition [21] was published to aid in portability and verifiability.

EUCLID [27], an offspring of PASCAL, is intended for the expression of systems programs that are to be verified. To permit stronger statements to be made about the properties of programs, EUCLID is yet more restrictive than PASCAL. For instance, the language guarantees that the aliasing problem can not arise, that is to say, two identifiers in the same scope can never refer to the same or overlapping storage areas. To effect this guarantee, EUCLID restricts the parameters that can be passed to a procedure. Hence, if we are given the declarations.

```
var A: array 1..100 of Integer;
var q: Integer;
procedure p(var x: Integer, var y:array 1..100 of Integer)
```

Euclid will not permit the function call

```
p(A[1],A)
```

since location A[1] would be known both as x and y[1] within p. However,

```
p(q,A)
```

is permitted.

3.3.3. Abstract data types

User-defined data types increased the programmer's ability to form his own data abstractions, but an important aspect of data abstraction was still missing. An implicit data type has a limited set of associated operations [22]. For example, one can perform arithmetic operations on integers, but not logical operations. However, a programmer who created a data abstraction by means of user-defined types could not limit the operations associated with the abstraction or restrict access to its representation. Programmers needed to be able to extend the concept of type to be a set of values together with the primitive operations that could be applied to these values. Additional support was necessary for users to enjoy the full advantages afforded to built-in abstractions in their user-defined abstractions.

The creators of SIMULA 67 [7, 8], generalizing the ALGOL notion of a procedure to apply to the problem of coroutines as used in simulation, developed a mechanism that was suitable not only for simulations but for the increased support of abstract data types. The SIMULA class contains both local data fields and operations to access these fields. Many instances of the class can be created, each identical but with its own private data. It is possible for the programmer to create a new, named type, to specify its structure by means of the class's local data, and to specify the operations associated with the type to be those provided by the class. Originally the language did not ensure that only class operations could access local data, but later versions enforce this restriction.

Other languages have developed facilities for linking the internal representation of a user-defined abstraction to the operations permissible on the representation and enforcing that only these operations access the representation. CLU clusters [33] and ALPHARD forms

[47] are two such examples. An example of a user-defined abstraction in CLU is a wordbag :

```
wordbag = cluster is
  create,    %create an empty bag
  insert  ,    %Insert an element
  print,     %print contents of bag
  rep = record [contents: wordtree, total: int];
create = proc () returns(cvt);
         return(rep${contents: wordtree$create (), total: 0});
         end create;
insert = proc (x: cvt, v:string);
         x.contents := wordtree$insert(x.contents,v);
         x.total := x.total + 1;
         end insert;
print =  proc (x: cvt, o:outstream);
         wordtree$print(x.contents,x.total,o);
         end print;
end wordbag;
```

Wordbag is the name of the abstraction, and the only operations permissible on *wordbags* are *create*, *insert*, and *print*. Only the cluster is allowed access to the internal representation of the abstraction by means of *cvt*, which stands for the object's internal representation.

Abstraction facilities have also influenced and been influenced by design methodology and verification considerations. The encapsulation of representation and operations eases verification, and suggests guidelines in the design process for the decomposition of programs [10, 13, 37].

4. The Abstraction of Control Constructs

The development of control abstraction does not parallel the evolution of programming languages as well as does the development of object abstractions. The procedure has been present since the first assembly languages, and many of the control constructs in use were first introduced when languages started abstracting away from the machine. A control scheme parallel to the concept of abstract data types has not emerged, unless it is the procedure mechanism that has been with us all along. However, there have been improvements developed in the proper support for different types of control abstraction.

4.1. What can be abstracted from control constructs?

What does it mean to form abstractions with respect to flow of control? An abstraction emphasizes the regularities among items, suppressing the less-important differences. It also indicates the intended use of an item, so that the item can be used without respect to the details of implementation.

The abstraction of control constructs reflects regularities in the flow of control. Since abstraction mechanisms indicate the general control flow, a reader can determine the order of execution of various parts of the program and how they interact without knowing their details. Control abstractions also guide the reader's focus of attention. Good programming style uses indentation to emphasize the regularity provided by control abstractions. When a reader encounters a control abstraction, he can assume that it groups activities that form a conceptual activity. Unless concurrent execution is involved, the flow of control will remain within the area demarcated by the scope of a control abstraction. For instance, the reader may safely assume that control will remain within a WHILE loop as long as the loop condition is satisfied, and that a procedure will complete execution before returning control to its caller.

The basic control-flow elements provided by computer hardware are sequencing (the linear order of instruction execution within the machine's memory) and branching, both conditional and unconditional. All control abstractions are built up from data manipulation and these flow-control elements.

4.2. Statement level control abstractions

Machine and assembly languages used the control elements of the hardware transparently. Programmers had the complete freedom of the machine, but with it shouldered the complete responsibility. Disciplined uses of the control elements arose to support such abstractions as procedures and iteration. For instance, the following conventions were adopted for procedure calls and parameter passing on the IBM 704*:

```
        TSX   ROUTINE,4
        PZE   address of parameter 1
        PZE   address of parameter 2
    .   .   .
        PZE   address of parameter n
```

For all procedure calls, index register 4 was expected to hold the address of the routine call. The called routine knew the number of parameters it expected, so it could calculate the return address. It would retrieve its parameters indirectly through the addresses that were stored in the code (note that such a technique makes the code non-reentrant). FORTRAN relieved the programmer of the necessity of remembering the details of the c convention by providing a subroutine call notation and generating the proper code. In FORTRAN, the programmer would make the above call by writing

```
CALL ROUTINE(parameter 1, parameter 2, ...  parameter n)
```

FORTRAN also provided improved control abstraction by its notation for arithmetic expressions [20], which relieved the programmer from register allocation responsibilities and concerns with order of evaluation. Instead of writing

```
LD  A
MUL B
ST TEMP
LD  C
MUL D
ADD TEMP
ST  F
```

---

*TSX means "Transfer and Set Index"; it stores the program counter into the indicated index register and then does a transfer. PZE stands for "Plus Zero", and is an assembler pseudo-op that generates a word containing only an address.

the programmer could write

```
F = A*B + C*D
```

Through DO loops, three-way conditional branching and subroutine mechanisms, FORTRAN provided a notation for the control disciplines established in assembly language.

ALGOL 60, designed with the goal of describing computational processes, introduced improved statement-level control abstractions. Block structure enabled the programmer to treat a group of statements as a single statement without the overhead of a procedure call. The IF-THEN-ELSE statement provided a convenient abstraction for conditional execution. In ALGOL, the programmer can use these control abstractions to write

```
IF X = Y THEN
   BEGIN
   X ← X-1;
   Y ← 0
   END
ELSE
   BEGIN
   X ← X+1;
   Y ← -1
   END;
```

while in FORTRAN, he would have to specify transfer of control explicitly:

```
      IF (X .NE. Y) GO TO 10
        X = X-1
        Y = 0
        GO TO 20
10    CONTINUE
        X = X+1
        Y = -1
20    CONTINUE
```

Looping mechanisms were provided by FOR-WHILE statements. Looping mechanisms are particularly useful abstractions because they relieve the programmer from the drudgery of decrementing counters and explicitly testing loop termination conditions, and also provide a standard repetition mechanism with which the programmer can become familiar. Errors involving one too many or too few iterations around a loop were much more common before such looping mechanisms entered languages.

The introduction of looping constructs and the desire to reduce or eliminate the use of GO

TO's in programming [9] led to the search for looping constructs with the flexibility to permit graceful loop exit but the control regularity to retain control decomposition advantages. Languages contain loop constructs with the exit tests at the beginning, middle, and end of the loop. WHILE loops continue as long as a condition holds, UNTIL loops as long as a condition does not hold. EXIT statements permit the programmer to exit a loop at any point; BREAK statements permit the programmer to advance to the next iteration of a loop at any point. MESA, EUCLID, CLU, and ALPHARD permit FOR loops to iterate over all the elements of a collection of arbitrary types, instead of just over a sequence of integers. For instance, in CLU the general form of the FOR statement is

```
for declarations in iterator do
    body
    end;
```

The user provides *iterators* with a collection definition; these iterators are used to yield the elements of the collection. Hence, if we defined a type *Set*, we could declare the iterator *ElementsOf* and use it:

```
for x: SetEl in ElementsOf(MySet) do
    process x
    end;
```

Language constructs such as the FOREACH statement in the SAIL language permit the programmer to abstract away from the order in which elements are processed, simply specifying that all the elements of a set are to be used. Programmers have been given more freedom to specify the iterative action and the termination condition.

More convenient means of grouping compound statements were proposed. For instance, BEGIN and END can be replaced by parentheses in BLISS [45] and ALGOL 68. Some languages introduce closing delimiters for control constructs. In ALGOL 68, looping structures repeat all statements between the delimiters DO and OD. FI is used to indicate the termination of an IF statement, eliminating the dangling ELSE problem [5]: in the following program fragment, it is unclear whether the ELSE belongs with the first IF or the second IF.

```
IF X = 0 THEN IF Y = 0 THEN  Z ← 0 ELSE Z ← 1
```

It can be disambiguated with the use of FI as a closing delimiter.

```
IF X = 0 THEN
  IF Y = 0 THEN
    Z - 0
  FI
ELSE Z ← 1
FI
```

4.3. Procedures as control constructs

A programmer can name an activity or group of actions and then consider them to be a single action. The language features that provide this ability are procedures (or functions or subroutines) and macros. The procedure, one of the first control abstraction mechanisms, is still the most common and useful. A procedure call specifies that control be passed to another program unit, which will return control to the calling point when it has finished its task. Parameter-passing mechanisms provide a means for the program units to explicitly exchange information.

Procedures permit the programmer the advantages of naming, discussed in section 3.1, in his use of control abstraction. The ability to group actions and refer to them by name permits the programmer to indicate the essential features of the group of actions. The abstraction advantages of this ability are greatest if the program units have minimal interaction and if the interactions they do possess are indicated explicitly. If this is the case, we can view a procedure as a machine instruction that affects only its parameters.

Procedures can indicate their direct effect on the state of a program by returning values. Languages place different restrictions on the amount and type of information that a procedure can return. In ALGOL 68, procedures always return values (although the type of the return value may be the VOID value EMPTY). In FORTRAN, a distinction is drawn between functions, which return a single value, and subroutines, which return no values. By use of the language features of constructors and extractors, MESA permits a procedure to return many values [14]. Procedures can also affect the program state indirectly by changing the value of global variables. Since such an indirect effect on the program state reduces the control abstraction advantages of a procedure, languages that permit multiple return values encourage the programmer in exploiting procedural control abstraction.

The advantages of procedures as control abstractions can be diminished by language features that permit side effects. Because of the possibility of parameter evaluation causing

side effects, it is impossible to write a procedure with two integer parameters called by name that will always swap the values of those parameters [12]. As a mild example, consider calling the procedure with subscripted parameters, e.g. SWAP(J,A[J]). The exchange will not always occur properly.

The use of global variables in procedures permits a procedure to affect the state of a program in ways that are not obvious from inspection of the procedure call. Certain parameter mechanisms, such as call by reference, permit aliasing; that is, they permit two different names within a name scope to refer to the same storage location. If aliasing can occur, the programmer can not reliably determine what abstract action a procedure provides by inspecting the procedure apart from its context in a program [20].

4.4. Exception handling

The limited focus of attention permitted by control abstractions and the nested call-return discipline of procedures aids the programmer in building up his program in an orderly fashion, but occasions arise in which such control disciplines unduly constrain the programmer. Conditions occur that cannot be handled locally but which must be brought to the attention of some other part of the program, such as the caller of a procedure. Such conditions are known as exceptional conditions or exceptions. For example, a procedure may discover that it has been passed invalid parameters and cannot perform its function; a storage allocator may run out of storage to allocate; an I/O device may encounter an end-of-file when trying to read data [16]. In demonstrating different mechanisms for handling exceptional conditions, we shall use the example of a storage allocator *GetBlock* that runs out of storage.

Many languages provide no special language assistance for dealing with exceptional conditions, forcing the programmer to employ standard language features for this purpose. A procedure can return a "return code" indicating success or the exception that arose during its call. The calling program must then test the return code explicitly at the conclusion of every operation, a clumsy and costly process that obscures the logic of the main program. In languages that permit procedures to return no more than one value, the programmer is obliged to return the function value indirectly, either through a global variable or by altering a parameter. In such a language we might declare our storage allocator:

```
TYPE RetVal = (success,OutOfStorage,badParameter);
PROCEDURE GetBlock (var ptr: BlockPtr): RetVal;
  . . .
```

and we could use it in the following way.

```
CASE GetBlock(newBlock) OF
  success: ...  continue normal processing
  OutOfStorage: ...  garbage collect and try to allocate block again
  badParameter: ...issue error message and abort
```

A program can deal with exceptional conditions by calling an exception-handling routine when it encounters an exception. This permits the handling routine to inspect the environment of the exception in its attempts to recover from it. Furthermore, all valid computations can be saved and need not be recomputed. However, if the handling routine cannot gracefully recover from the exception, control must still return to the exception site, from which it may be difficult to proceed in any reasonable manner. Hence, our storage allocator might include the following sequence:

```
PROCEDURE GetBLock: BlockPtr;
  BEGIN
  . . .
  IF OutOfStorage THEN
    OutOfStorage ← ¬GarbageCollection();   !garbage collection
  IF OutOfStorage THEN                     !returns true if more
    Abort;                                 !storage was obtained
  . . .
  END
```

Languages that support procedure variables, like ALGOL 68, provide more flexibility in the assignment of different handlers to an exception condition in different environments. A procedure variable is associated with each condition, and the procedure referenced by the variable is called when the condition arises. A subprogram can create its own handler for a condition by declaring a local variable of the same name and assigning its handler to it. The scope rules of block structured languages ensure that the handler most recently created will be invoked. An environment arranges to catch only those conditions it thinks it can handle. Consider procedures *Proc1* and *Proc2*, both of which call the storage allocator; furthermore, *Proc1* calls *Proc2*.

```
PROCEDURE GetBlock: BlockPtr;
  BEGIN
  . . .
  IF OutOfStorage THEN OutOfStorage ← ¬GetMoreStorage();
  . . .
  END;

PROCEDURE Proc1;
  BEGIN
  . . .
  GetMoreStorage ← GetStorageFromSystem;
  newBlock ← GetBlock();
  . . .
  Proc2();
  . . .
  END;

PROCEDURE Proc2;
  BEGIN
  . . .
  GetMoreStorage ← GarbageCollection;
  myBlk ← GetBlock();
  . . .
  END;
```

If *GetBlock* runs out of storage when called from *Proc2*, it will call *GarbageCollection*; if it runs out of storage when called from *Proc1*, it will call *GetStorageFromSystem*.

Another approach to exception handling is to transfer control to a handling routine by means of a GOTO, overlaying the current computation. This solves the problem of how to get rid of the environment in which the exception occurred, with the condition handler deciding where to resume computation. However, it is impossible to resume from the point of the exception if the handler finds it not to be fatal. Because there is no detailed information about individual exception occurrences, it is difficult to do anything but treat all occurrences uniformly. One cannot determine whether one instance of an end-of-file was expected and can be ignored, while another is a fatal error. Our storage allocator might include the statements

```
IF OutOfStorage THEN GOTO NoStorageExit;
. . .
NoStorageExit: PrintError("Out of Storage"); Abort;
```

PL/1, the first language to attempt to provide an explicit language mechanism for dealing

with exceptional conditions, uses such an approach. Conditions can be declared, and handlers are explicitly specified and associated with conditions dynamically by means of ON statements. Control transfers are achieved by means of non-local GOTOs, and parameter passing to exception handlers is not supported; this hampers the effectiveness of this mechanism for dealing with many conditions.

BLISS provides another, more sophisticated variant of this second method of condition handling with the more flexible SIGNAL and ENABLE mechanism. When an exceptional condition is encountered, a signal is raised by name. The signal propagates backwards through the call stack, looking for the most recent block or procedure that executed an ENABLE of the named condition. When it finds that ENABLE, it invokes the associated condition-handling routine. Upon completion of the handler, the block enabling the condition is terminated. The SIGNAL mechanism permits the condition handling routine to be specified dynamically, and it allows the exceptional condition to be handled by the part of the program that can best deal with it. However, it is impossible to return to the site of the condition if the handler can correct the condition. The condition is considered fatal by all calls until one is found that provides a handler. With this mechanism we could program *Proc1* and *Proc2* as follows:

```
PROCEDURE GetBLock: BlockPtr;
  BEGIN
  .  .  .
  IF OutOfStorage THEN SIGNAL NoMoreStorage;
  .  .  .
  END;
```

```
PROCEDURE Proc1;
  BEGIN ENABLE
          NoMoreStorage:
                BEGIN
                IF GetStorageFromSystem THEN
                   Proc1
                ELSE Abort
                END;
  . . .
  newBlk ← GetBlock;
  . . .
  Proc2(ENABLE                    !call Proc2 from within Proc1
          NoMoreStorage:
             BEGIN
             IF GarbageCollection THEN
                Proc2
             ELSE Abort
             END);
. . .
END;
```

If the storage allocator runs out of storage when called from this invocation of *Proc2*, the invocation of *Proc2* will catch the signal, call *GarbageCollection* and try again if more storage was obtained. If the storage allocator runs out of memory when called from *Proc1*, the same action is taken but *GetStorageFromSystem* is called instead of *GarbageCollection*.

A compromise between the method of handling an exception in its own environment and the method of transferring to another recovery environment is provided by the exception-handling mechanism in MESA [15]. It considers a signal to be a procedure call on the handler, except that the call binding is dynamically determined by the execution history, rather than statically by the lexicographic structure of the program. Parameters can be passed and returned, and the handler has the option of returning control to the condition site, permitting the signal to continue propagating up the call stack, or terminating the block it is associated with and continuing execution after the block. With this mechanism we would program *Proc1* and *Proc2* as follows:

```
PROCEDURE GetBLock: BlockPtr;
  BEGIN
  . . .
  IF OutOfStorage THEN SIGNAL NoMoreStorage;
  . . .
  END;
```

```
PROCEDURE Proc1;
  BEGIN
  BEGIN  !NoMoreStorage:
                BEGIN
                IF GetStorageFromSystem THEN
                   RESUME
                ELSE Abort
                END;
  .  .  .
  newBlk ← GetBlock;
  .  .  .
  Proc2(!NoMoreStorage:          !call Proc2 from within Proc1
             BEGIN
             IF GarbageCollection THEN
                RESUME
             END);
.  .  .
END;
```

If the storage allocator runs out of storage during this invocation of *Proc2*, the signal will be caught and *GarbageCollection* called. If more storage is found, we pick up from where we left off in the storage allocator; if storage is not found, the signal continues up the call stack and is taken by *Proc1*. It calls *GetStorageFromSystem*; if more storage has been obtained, we pick up where we left off in the allocator. If we still haven't obtained enough storage, we abort.

One drawback to this mechanism is that handlers must be located in the current nest of procedure calls. It is not possible, for instance, to invoke a handler associated with a data abstraction for all conditions associated with the abstraction.

Debates continue over the appropriate control path and context to use in handling such conditions. At present, languages provide few features for the support of exceptional condition handling. Levin [30] and Goodenough [16] survey and discuss techniques and language features for dealing with exceptional conditions and proposes additional solutions.

4.5. Effect of inappropriate control abstractions

The lack of appropriate and useful control abstractions has hindered certain control constructs from entering higher level languages. The hope of exploiting the concept of parallelism has enticed computer scientists for decades, but has met with little practical

success because of the lack of a proper abstraction for incorporating parallelism into programming languages.

One of the few languages to explicitly support parallel execution is ALGOL 68, which provides the PARBEGIN and PAREND constructs to give explicit indications of parallelism. This is a primitive tool for supporting parallelism, however. If a parallel activity begins in a routine, it must finish within the routine; parallel statements can neither take parameters nor return values; synchronization must take place by means of explicit semaphores [18]. We see that the programmer is forced to deal with the details of discovering opportunities for parallelism and controlling the interactions produced by parallel execution. The language offers no assistance in discovering and developing such opportunities. Hence it provides none of the guidance and regularity that a useful abstraction should. A proposal for an additional language facility that provides a more appropriate parallelism abstraction is presented in [18].

A more successful incorporation of parallelism is achieved by APL. APL provides powerful array-manipulation facilities that permit a high degree of parallelism in their implementations. For instance, when two arrays are added, all array elements can be added in parallel. The programmer, however, deals with the array abstractions without any need for considering their implementations. Hence APL has a much more usable control abstraction for parallelism.

Another related control construct that has floundered for lack of an appropriate abstraction is the coroutine. Coroutines imitate the interleaved execution of parallel processes, but deal with synchronization between programs more explicitly and in a simpler manner.

Coroutines are symmetric subroutines. One coroutine calls another, preserving its execution state just as a procedure does when it calls another procedure. However, a called procedure "returns", eliminating its local environment and all trace of its execution. A coroutine can, instead of returning, relinquish control by calling another coroutine. Every time a coroutine is called, it resumes execution wherever it left off when it last relinquished control.

Consider an *Edit* coroutine: it accepts input that consists of text and editing characters such as character, word and line deletions; it produces the edited lines as output. Any coroutine that needs edited lines would pass control to *Edit*. It would read characters, edit the line until it came to a line feed, and then pass the line and control to the calling coroutine.

Edit appears to call a procedure to process edited lines. The other coroutine appears to call a procedure to produce lines.

Language problems arise in creating and starting coroutines and in conveniently and flexibly specifying to whom a coroutine is passing control. In some cases, a coroutine needs to explicitly specify which coroutine it is calling; in other cases, it may need to pass control to whichever coroutine called it. It may be desirable that one coroutine be responsible for scheduling the others. If coroutines can pass parameters, these considerations become yet more difficult. Krutar [26] describes an efficient and flexible coroutine environment, and argues for their use in system construction.

Few languages provide constructs that permit the programmer to specify such interactions and guide him in the use of this tool. CLU's iterators are a constrained form of coroutine that are used only within FOR loops [33]. SIMULA 67 provides a coroutine abstraction [8] particularly useful in simulation with a standard language extension that provides automatic scheduling to imitate the passage of time within the simulation. MESA also provides support for a simulation abstraction that permits parameter passing and flexibility in passage of control.

5. Conclusion

We have outlined how the success and usefulness of programming languages has been intimately tied to their support of abstraction. This support has evolved as language designers have learned how to provide more effective assistance to the programmer.

The first higher-level languages provided built-in data abstractions, procedure mechanisms, and control abstraction facilities of varying degrees of sophistication and effectiveness. The advantages of this use of abstraction were negated by program languages becoming too baroque and providing too many such elements. As language designers recognized the need for meta-abstraction mechanisms to manage the overwhelming amount of detail, later language designs were simplified, but the tools were provided in the language for the programmer to construct his own abstractions.

Acknowledgements

I wish to thank my advisor, Nico Habermann, for valuable guidance in developing the ideas in this paper. Additionally, my thanks go to Brian Reid, for his help in devising examples, for his knowledge of early languages and machines, and for his assistance in the organization and style of this paper.

This document was produced using the SCRIBE Document Production System.

References

[1] ---, ASA Standard FORTRAN. *Communications of the ACM (October 1974).*

[2] ---, ASA Standard Cobol. Codasyl COBOL Journal of Development. National Bureau of Standards, Handbook 106 (1968).

[3] ---, ALGOL 68 User Manual. Carnegie Mellon University Computer Science Department Technical Report (1977).

[4] J.W. Backus, R.J. Beeber, S. Best, R. Goldberg, L.M. Haibt, H.L. Herrick,. R.A. Nelson, D. Sayre, P.B. Sheridan, H. Stern, I. Ziller, R.A. Hughes, and R. Nutt, The FORTRAN Automatic Coding System. Proceedings of the Western Joint Computer Conference (1957), 188-198.

An enlightening display of the goals and efforts of one of the first higher-level language and compiler efforts. Motivations for the project are discussed, the language defined, and the translator described a. great length. Heavy emphasis on optimization.

[5] P.W. Abrahams, A Final Solution to the Dangling else of A[lgol 60 and Related Languages. *Communications of the ACM* 9,9 (September 1966), 679-682.

Defines the *dangling else* problem, discusses solutions adopted by various languages and proposes syntax equations to eliminate the ambiguities.

[6] P. Brinch Hansen, The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering 1, 2* (June 1975), 199-207.

Introduces Concurrent Pascal, a language for concurrent programming. Extends Pascal with modules called processes, monitors and classes, attempting to combine module use for abstraction with module use for synchronization.

[7] Dahl, Myhrhaug, and Nygaard, The SIMULA 67 Common Base Language. Norwegian Computing Centre, Oslo (1968).

SIMULA 67 language definition.

[8] O.-J. Dahl and C.A.R. Hoare, Hierarchical Program Structure. In Structured Programming, Dahl, Dijkstra, and Hoare, Academic Press, London, 1972, 175-220.

Discusses features of SIMULA 67 in light of developing programming methodologies. Describes the use of SIMULA in modeling data abstractions, and discusses the use of coroutines in the language.

[9] E.W. Dijkstra, GOTO Statement Considered Harmful. *Communications of the ACM 11,3* (March 1968), 147-148.

Landmark paper presenting the benefits of control structure and the difficulties caused by the GOTO statement.

[10] E.W. Dijkstra, Notes on Structured Programming. In *Structured Programming*, Dahl, Dijkstra, and Hoare, Academic Press, London, 1972, 1-82.

Introduced programming design methodology to deal with the mounting complexity of computer programs. Proposes enumeration, induction and abstraction as the basic elements of program design. Marks a turning point in the awareness of the use of abstraction in programming.

[11] E.W. Dijkstra, The Structure of the "THE" Multiprogramming System. *Communications of the ACM 11, 5* (May 1968), 341-346.

Introduces the use of strictly hierarchical design as a tool in writing large software systems.

[12] A.C. Fleck, On the Impossibility of Content Exchange through the By-Name Parameter Transmission Mechanism. *ACM SigPlan 11, 11* (November 1976), 38-41.

Proof of the impossibility of writing a swap routine using call by name.

[13] L. Flon, Program Design with Abstract Data Types, *Carnegie Mellon Department of Computer Science.*

Explores the use of abstract data types as a modularization and structuring technique in the design of programs. Emphasis on design aspects of data abstraction. Well presented, with many examples.

[14] C.M. Geschke and J.G. Mitchell, On the Problem of Uniform References to Data Structures. *IEEE Transactions on Software Engineering, SE-1, 2* (June 1975), 207-219.

Develops notation for allowing a program to operate on a data object in a manner independent of its underlying representation. Describes many features of the MESA language.

[15] C.M. Geschke, J.H. Morris and E.H. Satterthwaite, Early Experience with Mesa. *Communications of the ACM 20,8* (August 1977), 540-553.

Discusses experience with strict type-checking and modularization within the Mesa system with an eye toward the interaction of language design and programming methodogy.

[16] J.B. Goodenough, Exception Handling: Issues and a Proposed Notation. *Communications of the ACM* (December 1975), 683-696.

Defines exception conditions, discusses requirements on exception handling language features, surveys existing approaches to exception handling and proposes new languages features to handle exception conditions. Good motivation for exception handling constructs and analysis of related issues, but only deals with language support up through PL/1.

[17] R. Griswold, J. Poage and I. Polonsky, *The Snobol4 Programming Language,* Prentice-Hall, 1971.

SNOBOL language definition and user manual. Excellent tutorial.

[18] P. Hibbard, Parallel Processing Facilities. In New Directions in Algorithmic Languages

1976, S.A. Schuman(ed), Institute De Techerche D'Informatique et D'Automatique, 1976, 1-7. Discusses some weaknesses with ALGOL 60's parallelism mechanisms and proposes eventual variables as an alternative.

[19] C.A.R. Hoare, Notes on Data Structuring. In *Structured Programming*, Dahl, Dijkstra, and Hoare, Academic Press, London, 1972, 83-174.

An excellent discussion of the usefulness of data abstraction in programming. Discusses the concept of type and proposes particular methods of data structuring.

[20] C.A.R. Hoare, Hints on Programming Language Design, *Stanford Computer Science Department Report No. CS-403.*

An excellent paper, presenting suggestions for designing languages that will most aid the programmer in the difficult aspects of programming, program design, documentation and debugging. Contains a brief annotated reading list.

[21] C.A.R. Hoare and N. Wirth, An Axiomatic Definition of the Programming Language Pascal. *Acta Informatica 2* (1973), 335-355.

A definition of PASCAL intended to be rigorous semantically as well as syntactically. Syntax diagrams used for syntax definitions and Hoare axioms used for semantic definitions.

[22] C.A.R. Hoare, Data Reliability. *ACM SigPlan 10,6* (October 1975), 528-533.

Discusses a method of data structuring and description, drawing parallels with methods of program structuring. evaluation.

[23] J.J. Horning, Some Desirable Properties of Data Abstraction Facilities. *ACM SigPlan 8,2* (March 1976), 60-62.

Draws parallels between procedures and abstract data types, particularly as regards the domain of computational structure.

[24] K.E. Iverson, *A Programming Language*, Wiley, 1962.

[25] R.G. Herriot, Towards the Ideal Programming Language. *Proceedings of an ACM Conference on Language Design for Reliable Software, ACM SigPlan 12,3* (March 1977), 57-62.

Suggestions for eliminating unnecessary redundancy and adding useful redundancy in programing languages, modeling suggestions after English language usage.

[26] R.A. Krutar, Flexors (Modification Mechanisms), Carnegie Mellon University Department of Computer Science.

Includes arguments for the use of coroutines in system building, and a description of an efficient, flexible coroutine environment. Difficult reading, particularly for those unfamiliar with the concepts discussed. Bibliography contains references to other sources for coroutines.

[27] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek, Report on the

Programming Language Euclid. *ACM SigPlan 12, 2* (February 1977).

Language definition presented in the style of the PASCAL report. EUCLID is interesting as a language designed to aid verification. While there is some discussion of the effect of this goal on the language design, most of the report deals strictly with language definition.

[28] C.P. Lecht, *The Programmer's PL/1*, McGraw-Hill, 1968.

A clear reference manual for PL/1. Not a tutorial.

[29] H.F. Ledgard and M.A. Marcotty, A Genealogy of Control Structures. *Communications of the ACM* (November 1975), 629-639.

Reviews theoretical results on control structures and explores their practical implications. Excellent survey of proposed control abstractions, sound theoretically, but also sound conclusions on the effectiv ness of control structures in programming task.

[30] R. Levin, Program Structures For Exceptional Condition Handling, Carnegie Mellon University Department of Computer Science.

Contains an excellent survey of exceptional condition handling mechanisms, for both sequential and parallel programs. A new mechanism is proposed and analyzed with regard to verifiability, uniformity, adequacy, and practicality. Lengthy but well written.

[31] B. Liskov, The Design of the Venus Operating System. *Communications of the ACM 15,3* (March 1972), 144-149.

Operating system constructed as a hierarchy of levels of abstraction. Hardware of the machine made more suitable to the system by means of microcode, creating more appropriate machine.

[32] B. Liskov and S. Zilles, Programming with Abstract Data Types. *ACM SigPlan Notices, 9* (September 1974), 50-59.

Argues for the need for abstract data type support in programming languages, and proposes clusters as such a mechanism.

[33] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, Abstraction Mechanisms in Clu. *Communications of the ACM 20,8* (August 1977), 564-576.

Provides introduction to CLU's abstraction mechanisms, supporting data, procedural and control abstraction. Well written and well motivated, explaining how the abstraction features of CLU support the programming task.

[34] J. McCarthy, Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM* (April 1960), 184-195.

Describes the formalism used by LISP for defining functions recursively. Discusses the implementation of a LISP system by means of list structures.

[35] G.J. Myers, *Software Reliability - Principles and Practices*, Wiley, 1976.

Chapter 15, Programming Languages and Reliability, contains examples of some non-intuitive results produced by PL/1's default and coercion rules.

[36] P. Naur(ed.), Revised Report on the Algorithmic Language Algol 60. *Communications of the ACM* (January 1963).

A landmark in language design and definition. First attempt to rigorously define language syntax via BNF. Draws distinction between the reference language, publication language, and hardware representation. Rigorous language definition makes report more useful as reference than tutorial. ALGOL 60's control structuring facilities greatly influenced future language design, and ALGOL implementation spurred compiler writing techniques.

[37] D.L. Parnas, Information Distribution Aspects of Design Methodology. *Proceedings of the IFIP Congress* (1971), 26-30.

[38] M. Shaw(ed.), IC Study Problems, Carnegie Mellon University Department of Computer Science.

A collection of programming problems for exploring languages and programming techniques. One problem has been used in this paper as an example of language influence on solution design.

[39] M. Shaw(ed.), IC Study Problems Solution Collection, Carnegie Mellon University Department of Computer Science.

The solutions to the problems given in the above technical report.

[40] P. Wegner, Programming Languages -- The First 25 Years. *IEEE Transactions on Computers, Vol. C-25, No. 12* (December 1976), 1207-1225.

An excellent overview of the principal concepts and programming languages developed since the 1950's. Each "milepost" is presented, with a discussion of what the significance of each is.

[41] N. Wirth, The Programming Language Pascal. *Acta Informatica 1* (1971), 35-63.

Defining report for PASCAL.

[42] N. Wirth, The design of a Pascal Compiler. *Software, Practice and Experience 1* (1971), 309-333.

[43] N. Wirth, *Systematic Programming: An Introduction*, Prentice-Hall, Inc., 1973.

Introduction to programming principles. Includes a brief description of the process of stepwise refinement.

[44] N. Wirth, Modula: A Language for Modular Multiprogramming. *Software -- Practice and Experience, 7 No. 1* (1977), 3-35.

Goals, philosophy and language definition of Modula.

[45] W.A. Wulf, D.B. Russell and A.N. Habermann, BLISS: A Language for Systems Programming. Communications of the ACM 14, 12 (December 1971), 780-790.

BLISS language definition with emphasis on features that support systems programming. Not a rigorous definition, but emphasizng goals and philosophy.

[46] W. Wulf and M. Shaw, Global Variables Considered Harmful. *ACM SigPlan 8,2* (February 1973), 28-34.

Discusses adverse effect of use of non-local variables on comprehensibility of programs.

[47] W.A. Wulf, R.L. London and M. Shaw, Abstraction and Verification in ALPHARD: Introduction to Language and Methodology, Carnegie Mellon University Department of Computer Science.

Discusses influence of ALPHARD's goals of support for development of well-structured programs and the formal verification of these programs. Presents current status of language, with references to other relevant sources.