REPORT ON

TEACHING ADA

(Revised)

SCIENCE APPLICATIONS, INC.

(1)

REPORT ON

TEACHING ADA

(Revised)

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION IS UNLIMITED (A)

SAI-81-313-WA

March 1980

Revised
December 1980

Russell J. Abbott
Associate Professor
Department of Computer Science
California State University
Northridge, California 91330

This report was supported by the Defense Advanced Research Projects Agency under DARPA Order No. 3456, Contract No. MDA903-80-C-0188, monitored by the Defense Supply Service, Washington, D.C. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency or the United States Government.

SAI
ATLANTA • ANN ARBOR • BOSTON • CHICAGO • CLEVELAND • DENVER • HUNTSVILLE • LA JOLLA
LITTLE ROCK • LOS ANGELES • SAN FRANCISCO • SANTA BARBARA • TUCSON • WASHINGTON

Science Applications, Inc.
1710 Goodridge Drive, McLean, Virginia 22102

Teaching Ada

Russell J. Abbott
Dept. of Computer Science
California State University
Northridge, Ca. 91330

Tel. 213-885-3398

Abstract

Two experiments to teach Ada in Computer Science classes are reported. It is concluded that Ada should not be taught simply as another programming language. Instead, Ada should be embedded within, and taught as a tool for, a coherent program design methodology. A continuation of the second experiment continues during the Fall '80 semester.

See P. 2

Report Organization

This report is divided into 5 sections:

1. Summary -

This section contains a summary of the lessons learned from the experiment and the conclusion to be drawn.

2. Ada -

This section discusses features of Ada from the point of teaching difficulty.

3. Discussion -

This section discusses the recommended approach to Ada and contrasts it with traditional approaches to teaching programming languages.

4. Course Summary -

This section contains more complete information on the courses taught during the Fall '79 and Spring '80 semesters. It concludes with a discussion of the course in progress, and

→ p. 3

5. Motivation

This section contains a 1 1/2 hour presentation which outlines a program design philosophy and provides a motivation for many of the constructs of Ada.

1. Summary

This section presents a summary of the lessons learned from an experimental project to teach Ada in (a) a first year Computer Science graduate class and (b) a Junior level class is program design. It also includes the conclusion drawn from the experiment.

1.1 Lessons Learned

Ada can be a difficult language to teach. There are 3 main problems.

a. Ada has a number of features which make it confusing. These should be given special attention by the instructor. (Section 2.1 lists some of these features.)

b. Ada is a formidable language. The presentation of Ada should simplify it for the novice user. (Section 2.2 presents an approach to such a simplification.)

c. Much of Ada is intended to support a program design philosophy rather than programming as such. These features are difficult for students to accept if presented simply as programming constructs. (Sections 3 and 5 discuss this issue.)

d. In order to teach Ada as a design language it is first necessary to develop a coherent approach to program design. No such approach is available in concrete enough terms to permit its teaching. (Sections 4.2.3 - 4.2.6 address this issue. The current semesters course begins to rectify the

situation.)

e. In order to teach program design, it is necessary to teach program specification. Design cannot reasonably be taught in a vacuum. If there is no specification, there can be no design -- for there is nothing to be designed if there are no requirements from which to work. So to do a complete job, it is necessary to develop a complete methodology leading, in an integrated way, from requirements to design. Ada would be most profitably taught within such a full program development framework. (Section 4.2.8 addresses this issue.)

1.2 Conclusion

Ada should not be taught as just another programming language. Instead, Ada should be presented within the context of a well developed program design methodology.

2. Ada

This section discusses features of Ada from the point of view of difficulty of teaching. While a number of features of Ada are reviewed, this section is not to be taken as an overall evaluation of the language. Only those aspects of Ada which are potential teaching trouble spots are mentioned. This discussion is based on preliminary Ada and does not reflect changes made in final Ada.

This section is divided into 2 main subsections:

2.1 Difficulties in teaching Ada

2.1.1 General Features

2.1.2 Specific Features

2.2 A view of Ada to facilitate teaching.

2.1 Difficulties in teaching Ada

2.1.1 General Features

Ada has the following general features which make its teaching more difficult.

a. Ada appears to the new user to be too big a language. Without guidance, the new user may feel lost in the large number of design concepts and programming features Ada offers. The new user needs help in determining which are central to the language and which can be put off until needed.

b. Ada programs tend to be quite lengthy (i.e. wordy), and for that reason, more difficult to get one's hands around.
c. Some of Ada's features are difficult to understand.
d. A supportive programming environment does not yet exist.

Ada is enough of a challenge to the new user by itself. The environment must not make it more difficult to work with Ada.

2.1.2 Specific Features

The following specific features make the teaching of Ada more difficult.

2.1.2.1 Multiple instances of similar things

Ada provides multiple ways of developing many instances of similar things.

a) Arrays of objects
b) Families of tasks and packages
c) Families of entries
d) Instantiations of generics
e) Multiple objects declared to be the same type

It may be confusing to the new user why there are different means for achieving similar results. In all of these cases, one is dealing with multiple instances of similar objects. Yet each of these features has its own individual properties which must be learned by the new user.

It might have been easier if packages and tasks were types, for example as in SIMULA, instead of objects. One would then use them by declaring objects to be of the desired package or task type. This would eliminate the need for families and for many of the uses of generics.

2.1.2.2 The overloading of language constructs.

Some of Ada's constructs are used for multiple purposes.

2.1.2.2.1 The keyword NEW

The keyword NEW is used for three functions :

a) Allocation: T:= NEW TREE ;
b) Derived types: TYPE A IS NEW INTEGER ;
c) Generics: PROCEDURE SWAP IS NEW EXCHANGE(CHARACTER);

These three uses of NEW occur at three different "times" in the processing of a program: run time, elaboration time and compile time respectively. The use of the same word in such different situations may tend to confuse the new user. One of the difficulties new users frequently have with programming languages in general is in keeping clear the distinction between the various phases in the processing of programs. The use of a single keyword for all three situations may compound the difficulty.

2.1.2.2.2 The keyword WHEN

The keyword WHEN is used for two functions :

a) CASE constructs
b) SELECT constructs

A confusion might arise in that one of these constructs is deterministic and the other is pseudo-non-deterministic.

2.1.2.2.3 The keyword RESTRICTED

The keyword RESTRICTED is used for two functions:

a) Visibility from inside a module to other modules;
b) The restricting of private data types from use in assignment or tests for equality.

These two functions are quite different. Again, the use of a single term may be confusing to new users.

2.1.2.2.4 The PACKAGE construct

The PACKAGE construct can be used for multiple purposes. All are valid uses and do not subvert the intent of the construct.

a) A replacement for named common;
b) Libraries of subroutinesi data types and constants;
c) Encapsulated data types;
d) Isolated modules (e.g. a "symbol table");
e) Co-routines;
f) An alternative to record-structured data types;
g) Abstract Machines;
h) Levels of abstraction.

It might have been better (at the risk of expanding the language still further) to identify some of those functions explicitly and provide individually named program objects for them.

2.1.2.2.5 The symbol =>

The symbol => is used for a number of functions:

a) CASE constructs
b) SELECT constructs
c) Object initialization
d) Parameter passing
e) Representation association

The first two of these are control functions; the second two are assignment functions; the last is a general association function. While the first four are also associations, their operational uses will probably predominate in people's thinking. The use of a single symbol may be confusing to new users.

2.1.2.2.6 The keywords EOB and USE

The keywords EOB and USE appear in the machine linkage statement: EOB ... USE... . They also have separate functions elsewhere in the language. Again, this may be a source of confusion. In this case, however, the difference between the functions is so great that the confusion is less likely to be severe.

2.1.2.2.7 Summary of Overloaded Constructs

The issues discussed in this subsection were all, presumably, deliberate design decisions. They may be good decisions; but they are also potential sources of confusion. If the same word or construct is used in multiple places, new users tend to believe that they are dealing with the same concept. In some of these cases, there may be a risk of significant confusion to new users.

2.1.2.3 Private Data types

Compilation requirements force private data types to be declared in the visible, specification portion of packages and tasks.

This is confusing in that it is not clear:

a) from the point of view of the package user, in what way the private type is hidden--since it is plainly visible;

b) from the point of view of the package specifier (i.e. the person who provides the requirements specification for the package), why it is necessary for him to specify those data types. As the person responsible for defining the functional capabilities of the private type, he is not supposed to need to be concerned about it's implementation. Yet the private type must be declared in the specification portion of the package.

2.1.2.4 Subtypes and derived types

It may be a source of confusion when one should use a subtype and when a derived type. It should be made clear that the only reason to define a subtype is to impose a range restriction on an existing type -- if it is, in fact, correct to make that claim. Derived types, in contrast, have much more significanct uses and are motivated by higher level methodological considerations. Yet the two type declarations appear so similar that this significant difference in their use may be lost on new users.

2.1.2.5 Range Expressions

It may be confusing in which contexts range expressions are permitted. They may be used:

a) In array declarations and elaborations.

b) In array slice operations.

c) In subtype, derived type and anonymous type declarations.

d) In object initialization.

e) In FOR statements.

f) In floating point and fixed point declarations.

g) In representation specifications.

h) In references to families.

It would be useful to develop a rule of thumb of the form "range expressions are permissible whenever ... ." Perhaps the blank is truly filled in by "whenever convenient." Expereince so far suggests that this might be the case. We have not had sufficient experience to enable us to make that assertion with complete confidence. See 2.1.2.10 for an apparent example of a situation in which a range expression and a type name for that range expression are not freely interchangeable.

2.1.2.6 Visibility: the keywords RESTRICTED and USE

The visibility features may be a source of confusion to new users. The keyword RESTRICTED serves both to

restrict visibility (from the normal ALGOL scope) and extend visibility. It may be confusing that Restricted visibility is in some ways greater that unRestricted visibility.

The USE clause may also cause trouble. Its utility is not as great as one might expect. The USE clause is to be applied only to modules--unlike PASCAL's WITH or SIMULA's INSPECT. So it is not a general "unwrapping" function. On the other hand, USE does not extend visibility in any way. Its advantage is minimal.

It might have been better to make the modules mentioned in a RESTRICTED list USEd by default, unless otherwise specified.

2.1.2.7 Programs and libraries

One of the most frequent questions new users ask about Ada is: what does a complete program look like?

It is a source of confusion that there are no objects called "programs". The library environment is a new concept and needs careful explaination to new users.

2.1.2.8 The term "overloaded"

The term "overloaded" was a poor choice. "Overloaded" connotes to most English speakers an unacceptable condition. An overloaded operator in Ada is not unacceptably burdened or ambiguous. The mechanism which supports overloading is very useful; the term itself is confusing.

2.1.2.9 Advanced programming features

Ada includes a number of advanced programming features which present some problem in teaching. While these constructs are important, they do take some effort to master.

2.1.2.9.1 Exceptions

The mechanism of declaring, raising and fielding exceptions and the associated flow of control are unfamiliar to most programmers.

2.1.2.9.2 Tasks

The creation and use of tasks are new concepts. New students must master concurrency, queues, asynchronicity,

pseudo-randomness and their appropriate uses.

2.1.2.10 Array declarations

It is unclear when, and to what extent, array bounds must be declared. For example:

```
Type vector is array (1..10) of integer;
v : vector;                    -- is apparently ok, but
Subtype index is integer range 1..10;
Type vector is array (index) of integer;
v : vector;                    -- is apparently not legal
                               -- because the array bounds
                               -- are not declared. Yet
type vector is
     Record
          size : constant index;
             v : array(1..size);
     End Record;
v : vector;                    -- is ok without declaring
                               -- a value for size, and
v := (10,(1..10 => 0));        -- is legal as is
v := (100,(1..100 => 0));      -- immediately afterward.
```

It is not clear just what does have to be known at elaboration time. (These examples are from E. Wegner, Programming in Ada.)

2.1.2.11 Elaboration time

Most programmers are familiar with the concepts of compile time and run time. In Ada, the concept of elaboration time becomes equally important. This is a new concept and may be a source of confusion to many new users.

2.1.2.12 Token disambiguation

If a token, for example GREEN, is declared in two types, for example color and traffic-light, it may be necessary to specify which GREEN is intended in certain situations. The notation created for that disambiguation is: color(GREEN). This functional notation is reasonable when considered as an extension of the notation for type conversion. As pure disambiguation, however, it is really quite awkward since no function is being applied. It might have been better to use the dot notation, color.GREEN, since this notation is used elsewhere for referential disambiguation.

2.1.2.13 The Generic Facilities

The facilities for generics are among the most confusing in Ada. Generics are neither pure macros nor are they pre-compiled modules. The notion that generics are compiled "as much as possible" creates difficulties. Non-local references in the body of a generic are bound in the scope where the generic is compiled. Yet parameters to

generics are bound in the scope where the generic is instantiated. And, of course, parameters to the resulting object -- if it has parameters -- are bound in the scope from which that object is referenced. In all three cases, objects are bound, but the scopes in which they are bound are different.

Secondly, generic declarations are ugly and hard to read. The attempt to pass function specifications as parameters just does not look intelligible.

Third, generics may not be powerful enough. Consider a generic integration function (wegner, p.157):

```
generic (function F(X: FLOAT) return FLOAT)
function INTEGRATE (LOW,HIGH:  FLOAT) return FLOAT is

...
```

If the user has defined:

```
type RADIAN is new FLOAT;
function SIN(x:RADIAN) return FLOAT is ;
```

will the compiler accept:

```
function INTEGRATESIN is new INTEGRATE(SIN);
```

To do so requires that it keep track of the derived type chain leading from FLOAT to RADIAN -- not a difficult job, but an added burdon. If the new function is accepted, will the parameters to it be required to be RADIANs or will arbitrary FLOATs be accepted?

Finally, generics are a weak form of a needed facility, see 2.1.2.15.

2.1.2.14 The Lack of Dynamic Types

Dynamic types could make Ada a much more flexible language for many purposes.

2.1.2.15 The Lack of Eval/Execute/Functional Parameters

For many purposes it is necessary to be able to perform one of the three facilities: passing functions dynamically as arguments, an "execute" facility or an "eval" facility. In the particular example examined in the Spring '80 class, it was very awkward not to be able to construct a procedure/function/command and then to execute it. See 4.2 for more details.

2.2 A view of Ada to facilitate teaching

Ada features may be divided into four categories:

a) ALGOL-like programming language features: the standard parts of the language.

b) Programmer conveniences: renaming, range operations, parameter passing by formal parameter name, etc.

c) Advanced computational concepts: exceptions, tasking, some of the object declaration features, etc.

d) Design motivated features: modules and associated mechanisms, the typing facilities.

These categories call for different teaching approaches.

2.2.1 The basic language

The ALGOL-like part of Ada may be taught in the standard way. It would probably be a good strategy to introduce experienced programmers to Ada through this part of the language so that they develop an initial sense of comfort and familiarity with it.

2.2.2 Programmer conveniences

These should not be ~~taught~~ at all. An attempt to teach these will only overload (in the bad sense) the students. The language will appear too big and with too many features. These programmer conveniences may be mentioned casually in

passing it the opportunity presents itself.

2.2.3 Advanced features

The advanced features should be taught semantically. (See section 3 for a discussion of teaching strategies). There are four steps:

a) A problem is presented which cannot be conveniently solved with standard features.

b) A discussion follows of the general form of the sort of programming construct needed for that problem.

c) That general-form programming construct is used to solve the problem in an informal, pseudo-dialect of Ada.

d) The actual Ada construct is introduced.

In this way the need for the construct is developed first. Second, its use is demonstrated and integrated. Only then is the student asked to learn the specific details of the construct.

Since the real-time, and other advanced features of Ada will be significant in determining its success or failure, these should be taught with great care.

2.2.4 Design features

The design features of Ada are very important to its successful use; they should be well taught.

The Ada design features should not be taught as programming constructs. In and of themselves, they do not contribute to the operational solution of a programming problem. The declaration of a derived type, for example, does not bring a program any closer to achieving its operational goals.

The most reasonable way to teach the design features of Ada is first to teach a design philosophy. Only then should one introduce the features of Ada which support that philosophy. Section 3 elaborates on this concept further. Section 5 is a 1 1/2 hour presentation which sets the stage for such an approach.

3. Discussion of Strategies for Teaching Programming

The teaching of programming has evolved through two stages. During the first stage, the teaching was syntax based; during the second stage, the teaching was semantics based. It is suggested that to do an adequate job of teaching Ada, a third approach is required. In this approach the teaching should be design based. This section provides a brief review of these three teaching strategies.

3.1 Syntax Based Teaching

The original approach to teaching programming tended to focus fairly heavily on the syntax of the programming language under discussion. Students were taught the format of the statements in the language--i.e. where the commas go, and the rules for forming variable names. It was implicitly assumed that once one could write syntactically valid statements in a language, it would be easy to write semantically meaningful programs. Of course this approach was not successful. One can teach a student the syntax of a language and he will still have no idea how he should use any particular construct or even why the construct was invented in the first place.

3.2 Semantics Based Teaching

In reaction to the failure of the syntax based approach, teachers developed a semantics based teaching approach. Using this approach one focusses explicitly on the semantic concepts available in the programming language. The syntax is brought in only as necessary and as a vehicle to express the semantics.

Using this approach, one would teach the semantics of looping, for example, and then discuss the DO statement (in Fortran) as the syntactic construct to be used to express looping. Similarly, one would discuss the notion of program variables (as, for example, names of boxes into which values can be put) and only secondarily discuss the rules concerning the spelling of variable names.

If one follows the semantics based approach to teaching programming, one can usually get across the main features of a programming language by considering the following aspects of the language.

a) Objects and Data Types

What are the pre-defined and programmer definable, program accessible, objects which the language makes available. If they are grouped into classes, what sorts of things are they and what do the classes represent. That

is, what are the data types, if any.

b) Operations

What operations can be performed on these objects. In what ways can they be manipulated by facilities built into the language. Normally these include at least the arithmetic operations and assignment.

c) Control Structures

What facilities does the language provide for organizing and controlling the operations.

d) Input/Output

What facilities does the language provide for communication with the outside world.

e) Other Features

What other facilities does the language provide. This category includes services provided by the language as well as language features like visibility rules and parameter passing modes.

3.3 Design Based Teaching

If one were to follow the semantics based approach to teaching Ada, one would find that areas a) Objects and e)

Other Features had grown considerably from even closely related languages like Pascal. Category c) Control Structures would also have grown, but to a much lesser extent. Compared to Pascal, for example :

a) Objects include an enormous amount of new information about data types. There is an entire new "time," elaboration time, in the processing of programs which becomes important.

c) Control structures include the new control structures for exception handling and tasking.

e) Other Features include a great deal of new information, including packages, scope rules and all the programmer conveniences defined in Ada.

In order to teach Ada successfully a higher level approach is needed. For just as syntax alone is not sufficient for the motivation and use of semantic constructs, the motivation and use of the Ada design constructs--especially the new information in categories a) and e)--is not generally evident from their semantics.

3.3.1 Teaching Strategy

Syntax was the initial focus of teaching programming because without correct syntax, one cannot write a program which will run. Because of that undeniable fact of life, when the teaching focus shifted to semantics, syntax still had to be accommodated. The accommodation is simple: it

is explained that there are syntactic constraints in writing programs and that they will be explained, but that they are on a lower level of importance than semantics.

A similar approach may be followed in a design based approach to teaching. It is explained that the specific semantics of Ada must be understood in order to write correct Ada programs, but that it is the design concepts that drive the use of the semantic constructs (in the same way as it is semantic concept which drive the use of syntactic constructs).

3.3.2 The Fundamental Design Constructs

What are the design concepts? There are a number of design qualities which are now generally considered important for programs. The two most inclusive and highest level are understandability and correctness. If a program posseses both of these qualities, most of the other design qualities follow.

In addition, it is just these two basic design qualities which motivate most of the Ada features not found in most other languages.

Basically, the teaching approach to Ada should start with understandability and correctness as basic requirements. These requirements should be used to derive design techniques such as strong typing, data type

encapsulation, modularity, information hiding, levels of abstraction, etc. Only as motivated by these design techniques should the specific features of Ada be presented as means for their implementation. Section 5 presents a talk which provides a design-based motivation for Ada constructs. A video tape presentation of the talk is available from the Aerospace Corporation, El Segundo, California. It was given there as a special forum on program design.

4. Report on Courses Offered Fall '79, Spring and Fall '80

4.1 First Year Graduate Course in Programming Langauge Semantics (Fall '79)

4.1.1 Course Organization

Ada was included in a first semester, graduate course in programming language semantics. Normally, the course explores operational models of several programming languages. This semester about half the time was spent discussing Ada. The SIGPLAN versions of the Preliminary Reference Manual and Rationale were used as texts.

The topics scheduled along with their associated sections of the manuals are shown in table 1. Only the first seven topics were actually covered in class. The teaching approach reported above had not been developed by the time the class was taught. The actual class was organized more along the semantic lines than along design lines.

Table 1

1. Types	P3,R4,R6
2. Statements/Operations	P4,P5,R3
3. Subroutines	P8,P6,R7,R9
4. Exceptions	P11,R11,R12
5. Packages	P7,P8,R8,R9
6. Multi-Tasking	P9,R4
7. Generic Types	P12,R13
8. Library System	P8,R9,R10
9. Input/Output	P14,R15
10. Numbers	P3,R5
11. Program	P1,P2,R14

Note : P - Preliminary Reference Manual

R - Rationale

Numbers shown are section numbers

4.1.2 Ada Interpreter in Ada

It was originally the intent of the course to teach Ada by developing an Ada interpreter written in Ada. That goal was not achieved.

The approach to that project was to presume the existence of a parsed Ada program available in the form of a parse tree. The job of the project was to write a program which would take that parse tree as input and execute the program it represents.

4.1.2.1 Representing the parse tree

To explore Ada "types" we examined the problem of defining a type NODE to be used in representing the parse tree.

Our approach to representing the parsed program tree was 1) to create an access type called NODE to represent each node; and 2) to follow the grammar more or less directly from the manual.

```
TYPE LEFT_HAND_SIDE_SYMBOL IS <All the left hand side
                                grammar symbols>

TYPE NODE IS ACCESS
    RECORD
        NODE_TYPE : CONSTANT  LEFT_HAND_SIDE_SYMBOL;
        CASE NODE_TYPE OF
          WHEN <LEFT_HAND_SIDE_SYMBOL> => <RIGHT_HAND_SIDE>;
                    ...          =>      ...
                    ...          =>      ...
        END CASE
    END RECORD
```

Where <RIGHT_HAND_SIDE> is represented as follows depending upon the form of the associated grammar rule. The following shows the various allowable forms of grammar rules.

1. Grammar rule A -> B C D ... E

   WHEN Clause:

```
WHEN A=> RECORD
                    B_PART : NODE(B) ;
                    C_PART : NODE(C) ;
                    D_PART : NODE(D) ;
                      .
                      .
                      .
                    E_PART : NODE(E);
          END_RECORD
```

2. Grammar rule A -> B | C | D

   WHEN Clause:

```
WHEN A => OFFSPRING : NODE(B | C | D)
```

3. Grammar rule A -> Terminal

   WHEN Clause:

```
WHEN A => LITERAL : STRING
```

4. Arbitrary repetition (indicated by '{' and '}' in the grammar).

   A => {B} is represented by

   THEN A => ARRAY(NATURAL) OF NODE(B)

   (Note that any constructs of the form {A...B} can be replaced by {C} and C -> A...B.)

5. Optional components (indicated by '[' and ']' in the grammar are always included in the node. (If the component does not exist in the actual situation, the value of the component is simply Null).

6. Key words are not included in the tree. (Their purpose is to permit parsing.)

7. Minor liberties are taken with the grammar to simplify the situation.

As an example, consider rule 5.4 in the grammar:

```
IF_STATEMENT ::=
        IF <CONDITION> THEN <SEQUENCE_OF_STATEMENTS>
            {ELSEIF <CONDITION> THEN
                <SEQUENCE_OF_STATEMENTS>}
            [ELSE <SEQUENCE_OF_STATEMENTS>]
        END_IF;
```

The node type representation is:

```
WHEN IF_STATEMENT =>
   IF_PARTS : ARRAY(NATURAL) OF
      RECORD
         CONDITION_PART : NODE(CONDITION);
            THEN_PART : NODE(SEQUENCE_OF_STATEMENTS);
      END_RECORD
```

It seemed clear after working with the grammar for a while that TYPE node could be defined. It also seemed clear that actually to do it would be a long and tedious job.

4.1.2.2 Representing the Run-time Environment

Next we attempted to define a "type" to represent general Ada objects to be kept in the execution environment. It turned out to be a difficult job. The type became so complex that we gave up.

4.1.2.3 Use of the Ada Rational

The remaining sessions were spent going through the Reference Manual and the Rational. The Rational was a excellent text for discussing both Ada in particular as well as issues in the design of programming languages in general.

4.1.2.4 Use of the AdaTE Facilities

Many attempts were made to use the AdaTE test translator. We lacked familiarity with it and with the various intermediary systems required to gain access to it. We were never able to make constructive use of it.

4.2 Junior Level Course in Program Design (Spring and Fall '80)

This was the appropriate course in which to teach Ada. Although the first trial was not perfect, it brought to light a number of points which should be incorporated in future courses.

4.2.1 Background to the Course

The intent of this course is to teach program design to students who have a reasonable facility with programming tools. It follows two years in which students learn the basics of programming. It has as prerequisites three Sophomore courses and three Freshman courses.

Freshman Requirements

Introduction to Algorithms

A first course in programming using a high level language.

Introduction to Computers

A first course in computer organization -- i.e. assembly language programming.

Data Structures

An introduction to data structures and their implementation.

Sophomore Requirements

Computer Organization

An introduction to computer organization and systems programming. Students write a simple assembler and an emulator for a PDP11 type

machine.

Concepts of Programming Languages

A survey of programming language concepts and facilities as provided by a variety of high level programming languages.

Introduction to File Organization

An introduction to the storage of large amounts of information outside the direct control of a program.

This background gives our Junior students two solid years of training. They have been exposed to most of the basic tools of programming. They are taught top down design and structured programming techniques as part of their normal programming courses. This background would qualify them for jobs as program implementers: given a description of a desired module (requiring up to about 500 lines of HOL code), they could design and produce a reasonable implementation.

These students have not had experience with large programs. The programs they build in developing their assembler/emulator grows during the semester into quite a big program. But the separate pieces are given to them one by one. They do not have to worry too much on their own about how they fit together.

The Program Design course in which Ada was used is intended to teach techniques for dealing with programs too large to keep in one's head all at once. This seems to be the perfect course in which Ada (as distinct, for example, from Pascal) is uniquely

valuable. Ada's Package construct is ideal for large program design. In addition, it is just this situation -- i.e. in which the program under consideration is too big to hold in one's mind all at once -- for which Ada was designed.

4.2.2 Course Entrance Test

Although the prerequisites to the course are fairly rigorous, it often is the case that students do not come fully prepared. In an attempt to calibrate the level of preparedness of the students, the following examination was given.

Students were given copies of Chapter 1 of [wegner]. They were told to read sections 1,2,4,5,6,8,9,10 and 11 from that chapter. They were told to write down any specific items they did not understand. They were told they would be quizzed on the material. The next week, they were told to hand in their list of questionable items. Then, with no assistance they were given the following quiz.

QUIZ

Answer the following questions. If any pertain to any of the items you have marked as not understandable, mark that question "deleted." You have 20 minutes. The quiz is open book.

```
Procedure Simple-add is
     x,y,z: integer;
Begin
     get(x);
     get(y);
     z := x+y;
     put(z);
End
```

1. The variables x,y and z are
   (a) local variables to the procedure;
   (b) formal parameters to the procedure;
   (c) global variables to the procedure;

2. The character "_" means
   (a) subtraction
   (b) nothing; it is part of the name

3. What does the construct "2 .. 10" in the following mean

```
For i in 2 .. 10 loop
```

   (a) 2 raised to the power 10
   (b) the range of integers from 2 to 10 inclusive
   (c) the range of real numbers from 2 to 10 inclusive.

4. In

Procedure Sort (a : in out vector) is

the words "in out" mean

(a) the vector a is not originally sorted

(b) the vector a may be accessed by the procedure and may also be changed by it

(c) the vector a may have components interchanged by the procedure.

5. The basic program unit in Ada is called a "program" and is generally of the form

```
Begin
     <declarations>
     <statements>
End
```

(a) True

(b) False

6. The formal parameters to a function may be declared in out and may be modified by the function to return additional information to the caller of the function. (True or False).

7. Consider:

```
Package Math_functions is
     function sin(x:real) return real
     function cos(x:real) return real
     function tan(x:real) return real
End
```

The above package specification gathers together the names of the functions sin, cos and tan. Their actual definitions are taken from the math library of math functions as part of the run-time support of the language. (True or False)

8. An operator is overloaded if

(a) it has more than one interpretation

(b) it has too many interpretations.

9. The relationship between a package specification and a

package body is

(a) the body defines the things that were declared in the specification

(b) the body uses the things that were declared in the specification (as a procedure body uses the parameters and local variables).

10. If a data type is declared **private** in a package specification, it is

(a) available for use by the user, but its components are not directly accessible

(b) not available for use by the user

(c) available for use by the user if given permission by the definer.

11. A task in Ada is

(a) a program unit which can run concurrently with other program units

(b) a well defined component of a large programming project which can be set off by itself using the stepwise refinement approach to program design

12. An **accept** statement permits a task to accept

(a) data from another task

(b) control from another task

(c) both of these

(d) neither of these

(d) all of the above

13. The purpose of the construct

```
Select
        When <condition> => accept ...
    or
        When <condition> => accept ...
End Select
```

(a) to enable a task to be available for calls from multiple users of the task

(b) to select the one (and only one) condition among all those indicated which is True and then to perform the accept statement associated with it.

The students were told that this test was not going to be used in determining their final grade. They should consider it as a readiness examination. The level of this course presumes that they are more or less capable of reading the assigend material on their own and determining what they did understand and what they did not understand. It turned out that the test was a fair predictor. The students who scored below 7 generally got C's or worse. All the A's in the course were earned by students who scored 10 or better. But some of the high performing students fell off later and one of the poorer scoring students did earn a B.

The results of the test were:

Number correct	Number of students
13	0
12	3
11	0
10	4
9	5
8	10
7	5
6	0
5	3
4	3
3	0
2	1
1	0
0	0

4.2.3 Intended audencies for courses in Ada.

There are probably three types of audiences for serious Ada courses -- i.e. courses which are more than just a brief executive overview of what Ada can do for a large system. These three audiences are: programmer managers, experienced programmers and novice programmers.

In at least the first two of these cases, it seems that the appropriate focus should be on the large scale program design issues: programmer managers should certainly be concerned about

program design; experienced programmers generally have design as their greatest weakness. Novice programmers, however, need more than design; they need the experience of writing concrete programs. Otherwise they don't develop a feeling for what programming is all about. But even in this case, the program design philosophy should infuse the more concrete programming ideas taught.

So the question becomes: how does Ada work in a course in Program Design?

4.2.4 Difficulties in Teaching Program Design

The primary difficulty in teaching program design is that there is no clear definition of what a proper program design should be. It is much easier to teach programming than program design in that one can at least run the program and see if the answer is right. Design is much more qualitative: one cannot give an automatic test to apply to a design.

In addition, we have no standards for design. We have developed standards for programs: single-entry/single-exit control structures, hierarchical module structure, "small" module size, strongly typed data object usage, etc. There are not similar standards for design. So the two basic problems one faces in teaching design are:

1) What is a program design?

I.e., how would you know one if you saw one?

2) What qualities characterize good program designs?

The first question is the most revealing. There is no agreement in the software community as to what one is producing when one does software design. There does seem to be agreement about levels of design. There are two: architectural or top level and detailed. There also seems to be agreement about a characteristic of detailed design: it can be turned into a program with a minimum of effort.

But the important question is about the architectural level of design. Here there is no consensus. It is interesting to note that the DoD has standards for program specifications (all program specifications must be written in a certain format) and for programs themselves (all programs must be written to certain formats). But there is no standard for design: anything apparently will do.

In Wegner's 869 page book, "Research Directions in Software Technology (Ed. by P. Wegner, MIT Press, 1979) there does not seem to be one definition of what a software design should be. In the software engineering literature there is little agreement about design. There are the notions of "Top Down Design," Structured Design," "Michael Jackson Design," "Parnas Design," and others. To a great extent, these are as much design techniques as design definitions. As techniques, one can infer a design definition implicit in them. However, their goals seem for the most part to be to find ways to produce code in an orderly manner. This is different from an explicit goal of producing a design itself as an end product. Most of the work in

program design, then, seems to take the word "design" as a verb (and hence attempts to define a process by which one designs and codes programs) rather than a noun (in which the design itself is a desired product). And, in fact, one hears a great deal about "program design methods" and a lot less about "program design standards."

4.2.5 what is a design?

A design is a description of the thing designed. A design shows the structure of the object. According to Webster's a design is: "a plan," "a delineation," "a preliminary sketch," "an underlying scheme that governs functioning, developing, unfolding," "the arrangement of elements that make up a structure or a work of art."

Software design is a description of the structure and functioning of software. What is necessary to show that? There are at least four sorts of structures relevant to software: control, data, interface and framework. A software design must delineate all four.

1) Control structure.

Control structure is the most familiar of these notions to those involved with software. The control structure of a piece of software is the paths which the processor(s) which perform(s) the software may take through it.

2) Data Structure.

The data structure of a piece of software is the ways in which

data is stored within it and the paths which data may follow through it.

3) Interface Structure.

The interface structure of a piece of software is the orgainization of the interactions between the software and elements external to it.

4) Framework.

Beyond these three sorts of "flow" structures, there is a fourth, and perhaps more important structure. That is the structure of the framework of concepts and definitions upon which all the others are based. It is this fourth level of structure -- the static, skeletal structure within which the others function -- that is at the heart of design.

It is the goal of this course to teach program design, where design is defined in the terms just discussed.

4.2.6 Lessons Learned about Teaching Design

1) Students have a hard time developing a feeling for what one wants in a design.

2) Students have a hard time producing a design in any sort of logical way. Often they seem to "guess," i.e. to write down something without an understanding of what it means.

3) Students have to be taught about the notion of a black box module.

4) Ada is a useful language to use for expressing software design.

5) It is necessary to teach system specification before teaching design. It is necessary to teach system requirements definition before teaching system specification.

6) It is very difficult to use Ada (or any formal language) without an easy-to-use processor.

4.2.7 Problem Assignments

4.2.7.1 First Assignment

The first assignment was to specify packages for the "pile," "tree," and "counter" objects introduced in the "programming methodology" talk (see Section 5.).

answers

```
Package COUNTER_PACKAGE is
     Type COUNTER is private;
     Procedure Increment(c: in out COUNTER);
     Procedure Reset(c: out COUNTER);
     Function Value(c: COUNTER) Return REAL;

Private Type COUNTER is new INTEGER;
End COUNTER_PACKAGE;

Package Tree_stuff is
     Type tree is private;
     Procedure tear_in_half(t: in tree, l,r: out tree);
```

```
        Procedure discard(t: in_out tree);
        Function is_leaf(t: tree) return BOOLEAN;
        Private type tree is access
            Record
                tree_or_leaf : (T,L);
                Case tree_or_leaf of
                    when T => L,R : tree
                    when L => A : ??????? data ???????
                end case
            end record
    end tree_stuff;

    Package Pile is
        use tree_stuff;
        type pile is private;
        function is_empty (p:pile) return boolean;
        procedure reset (p: in_out pile);
        procedure put_on_pile (t: in tree, p: in_out pile);
        procedure pull_off_pile (t: out tree, p: in_out pile);
        Private type stack is
            array(1..100) of tree;
    end pile;
```

Although the assignment was for the students to give only the specifications for the three packages, the bodies were worked out in class. The class seemed much more comfortable if they could see how the bodies would work. Otherwise, they felt a lack of concreteness. (It should be noted that although the example

in section 5 made use of a "pile" data type, this solution defines the single pile required as an encapsulated data object.)

```
Package_body COUNTER_PACKAGE is
     Procedure Increment (c: in_out COUNTER) is
          begin c := c+1 end;
     Procedure RESET (c: in_out COUNTER) is
          Begin c := 0 end;
     Function VALUE (c: COUNTER) return REAL is
          Begin Return c end;
end Counter_Package;

Package_Body tree_stuff is
     function is_leaf(t:tree) return BOOLEAN is
          t.tree_or_leaf = L;
     Procedure trear_in_half (t: in_out tree, l,r: out tree)
     is
          Begin
               Assert not is_leaf(t);
               l := t.l;
               r := t.r;
          end tear _in_half;
     procedure discard(t: in_out tree) is
          t := null;
end tree_stuff;

Package_body pile is
     function is_empty (p:pile) return boolean is
          top = 0;
```

```
procedure put_on_pile (t: in tree) is
     begin
     top := top + 1;
     P(top) := t;
     t := null;
     end;

procedure pull_off_pile (t: out tree) is
     begin
     assert top > 0;
     t := p(top);
     top := top - 1;
     end;

procedure reset is top := 0;
     top : integer_range 0 .. 100 := 0;
end pile;
```

4.2.7.2 The Second Assignment

The next assignment was to do the same job (i.e. create packages) for another problem. This problem was one that one of the other instructors uses in his version of the class. He has written a set of notes for the class. [Gilbert] Those notes have been used as a class text, so we used some of the problems from them.

The_Top_Ten

"Rocky Racoon Records is trying to improve record sales.

For this purpose, the company has put together a list of names and addresses, and has hired the Random Sampling Co. to interview the people on this list every month. People on the list will be grouped into four categories, according to their age (age < 20 or age > 19) and sex (male or female), so that the company can examine the various groups who buy their records.

"Each person on the list will be asked to state which records of the Top Forty he or she would rank 1st, 2nd, 3rd, ... 9th, 10th. This information together with the interviewee's name and address, will be punched on cards for input into a program.

"The program is desired which will process these cards and print out various lists:

(a) to find out which records are popular

(1) a list of the Top 40 titles, in alphabetical order, with the number of times the title was mentioned.

(2) a list of the 10 most popular titles, in order of their popularity.

(b) to find out who are the best record customers

(3) a list of all people who mentioned at least 5 out of the titles in list 2

(4) four separate lists (one for each of the four categories), each list naming all interviewees in the category who rank 1st one of three titles most popular with people in the category."

Again, the solution was found by first generating an intuitive solution -- including a data flow graph. Then the objects and operations in the intuitive solution were to be mapped onto Ada packages.

It turned out that all that was needed was a single package with the following types and operations.

Types (Private)

Title, Response, Person, Table_of_titles, Table_of_People, Table_of_Responses

Operations

```
Sort_titles(In_OUT Table_of_Titles);
Add_to_Mentions( in Responses, In_out table_of_titles);
Calculate_popularity( in table_of_responses,
                      in_out table_of_titles);
pick_best_choosers(in title-table,
                   in response-table,
                   out people_table);
pick_most_representative_of_category(in title_table,
                                     in response_table,
                                     out people_table);
```

4.2.7.3 Exercise and Quiz on data structures

It turned out that the students in the class had a much poorer background in defining data structures than they should have had. We spent a good part of the next few classes working on data structures and on Ada facilities for their expression.

Following that, this quiz tested what they had learned.

QUIZ

This quiz represents a package to keep track of information about students, teachers and courses. The three data types are defined:

```
type student is access
     record
          name : string;
          current_gpa: gpa;
          units_attempted, units _passed: units;
          updated :semester;
          courses: array (natural) of course;
               -- all courses ever attempted
     end_record;

type course is access
     record
          name: ticket;
            -- a 5 digit ticket number
          teacher; instructor;
          offered: semester;

                -- all sections ever offered are kept in
          the data base
          students: array (natural) of
               record
                    person: student;
```

```
                    evaluation: grade
                      -- entered at the end
                      -- of the course
                end_record;
        end_record

type instructor is access
        record
            name: string;
            title: rank;
            courses: array (natural) of course;
        end_record

package school_info is
        type student is private
        type instructor is private;
        type course is private;
        type ticket is private;
        type semester is private;
        type rank is private;

        function find(name:string) return student;
        function find(name: string) return instructor;
        function find(nbr: ticket) return course;

            procedure update_student(person: student,
                time: semester);
private
        type rank is ...;
```

```
        type ticket is ... ;
        type semester is ... ;
    end school_info

    package body school_info is

        type node is access
            record
                tree_type: constant (student, instructor,
                courses);
                case tree_type of
                    when students => value: students;
                    when instructors => value: instructor;
                    when courses => value: course;
                end case
                left, right: node := null;
            end record

            student_root: node := null"
            instructor_root: node := null;
            course_root: node := null;

            function find is new gen_find(string,student);
            function find is New gen_find(string, instructor);
            function find is new
                        gen_find(ticket, course);

            procedure update (person: student, time:semester)
            is ... ;
    end school_info;
```

Questions

1. The types student, course, and instructor could just as well have been defined as non-access types (true or false: explain your answer).

2. Complete the definitions of the type ticker. It should be with a subtype of integer or a derived type. Explain why you are making the choice you do.

3. Create a definition for the type rank (in the instructor type). It should be an enumeration type which can take on values of: Asst_prof, Assoc_Prof and Full_prof.

4. The package body contains three trees similar to the symbol table tree in the the symbol table package in [Wegner, p. 143]. In shcool_info there are trees for students, instructors and courses. A single node type is defined with a variant record structure.

Rewrite the function find [Wegner, p.143] so that it takes a student name as input and returns the student if found. If not found, the value null should be returned.

5. Since there are three trees with identical structures, it seems reasonable to use the same find function on each. But since Ada enforces strong typing, problems arise:

a) the input to two of the find functions are strings (instructor name and student name); the input to the other find function is a ticket number (for course).

(b) the results returned are of three different types: student, instructor and course.

This is an ideal situation in which to create a **generic** function.

Modify the answer you gave for (4) so that it is a generic function. The two generic parameters are (a) the type of the input to the function and (b) the type of the output. Name the generic fucntion gen_find so that the find functions as defined in the package body are correct.

6. Complete the **Procedure** update. The inputs are as shown.

Processing:

1. Look through the list of courses associated with the given student.

2. Select those whose "offered" copmpnent matches the tiem parameter.

3. Update the student's units_attempted, units_passed and gpa by each such course.

4. Update the student's "updated" component to be the inut parameter.

Which additional type parameters have to be completed? Show how you would define them.

Answers

1. **False**

Access types are required to permit objects to point to other objects. Students, courses and instructor all point ot one another. If tne types were not access, each of these types would have to be embedded within the definitions of some of the other types. E.g., courses would have to be embedded within students. But students would also have to be embedded within courses.

2. Derived type

```
type ticket is new integer range 10000 .. 99999;
```

Don't want to permit ticket numbers to be confused with other sorts of integers.

3. `type rank is (Asst_prof, Assoc_prof, Full_prof);`

4. Need to change program in the book to compare

```
n < n.value.name
```

This tested the understanding of records and access types.

5.
```
Generic (Type S,R; Function "<"(u,v:s) return boolean)
function gen_find(name:  S, root:  node) return R is
    n:node := root_type;
    begin
    end;
```

6. `Procedure` Update ...

4.2.8 Requirements and Specifications

After reviewing as much of Ada as was indicated, it seemed appropriate to back up and look at the problem again. (The problem is how to design programs.) The design problem is, of course, embedded in a larger problem: the program life cycle. We spent some time going over the standard waterfall chart.

In order to do a reasonable job of program development, four tasks are required:

1) define the user's needs,
2) specify the computer system which will fill those needs,
3) design that system,
4) code that system.

The explicit goal of the course is to teach techniques for step 3. But there really is no way to do step 3 in a vacuum. There is no way to design something without knowing what it is that has to be designed.

Requirements

We spent some time figuring out what user needs ("requirements") were. We used as an example problem, a travel agency system. The system should be able to set up trips for travelers. It would have to deal with interconnection, travel preferences (e.g. meal types, time contraints, etc.) fare calculations, etc.

We spent some time investigating a requirements statement for the problem. It was a very useful exercies for the students

to see how much information has to be developed.

We developed a general framework for a document in which to express requirements. Briefly it included:

0. Document Overview and Conventions
1. Operational Context: purpose, goals and world model.
2. Data Requirements
3. Functional Requirements
4. Interface requirements
5. Human Factors
6. Performance: timing, sizing, loading, accuracy
7. Other Constraints

We didn't include a section on standards because the standards are not really a part of the program specific information. All systems should be designed to essentially the same design and programming standards. If necessary, there should be a general reference standard which can be invoked.

Specifications

Since requirements were fairly far from the design problem, we went on to discuss specifications. For that we used the same document format as for the requirements but included in it system specifications rather than user goals. The essential differences between the requirements document and the specification documents are:

1) The requirements document is written from the point of view of the user. It speaks in user domain terms, in English,

about user domain issues; discusses data and operations with respect to the user organization; and has as its overriding goal the expression of the goals of the user and his organization.

2) The formal specification is written from the point of view of a independent outside party. It speaks in mathematical terms and in predicate logic notation whenever possible; discusses data and operations in formal terms and with respect to precisely defined interfaces; and has as its overriding goal the expression of a precise description -- i.e. specification -- of a system which will satisfy the goals in the requirements document.

4.2.9 Final Exercise

One more exercise was given in the course. It was a variant on the matchmaker problem. We discussed the algorithmic problems involved and then concentrated on the user needs -- i.e. we considered what an actual matchmaker business would like to have, rather than worrying too much about the computational complexity of the general problem.

4.2.10 Final Project

For the final project, the students were required to design a problem for which Ada is not well suited. The problem is to build one or more packages which an application programmer could use in developing specific application programs.

we decided to design a general graph handler to model possible user interaction paths. It was also necessary to define a general input type checking module. Here Ada was not easy to use unless one wanted to build generic modules. In addition, we wanted to build facilities to construct calls to unspecified input processors. Again, Ada was difficult to use because the details about the input processors would not be available until after the program was written. That information would be provided by the application programmers. Ada is not very suitable in that one cannot build procedure calls dynamically and then execute them. The problem was worthwhile, however, since it did lead to some useful packages. For those situations in which Ada was weak, we specified augmented capabilities which would make the problem easier. That too was a useful exercise in specification.

The general outline of the design has five packages.

Terminal Screen. This package is a specification of the user terminal screen. The actual screen is assumed implemented in hardware.

Terminal Keyboard. This package specifies the user keyboard.

Terminal Handler. This package handles the user input. It groups together input characters according to formats passed to it from the graph handler. It accumulates a number of "items" in specified formats and within specified ranges. Each such collection of items corresponds to a single

"screenful" of information -- e.g. perhaps a form or a collection of responses to a menu. If the user input does not match the required formats, the terminal handler interacts with the user to help him correct his mistakes. When the user input is complete, the terminal handler hands the input collection over to the graph handler.

Graph handler. This package is the central controlling part of the system. It follows the user around the paths provided by the application programmer. Each application provides a graph which indicates the possible user interactions. Each node of the graph is associated with a single user interaction sub-session handled by the terminal handler. Each such node has a collection of items associated with it. These items, and whatever prompting material goes along with them, is passed to the terminal handler for display to the user and for data collection. In many cases, a node is a menu requiring a selection by the user. In other cases, a node is a form to be filled in by the user. In all cases, the graph handler receives the input back from the terminal handler. It then packages that information as arguments to a procedure call associated with the node and calls that procedure.

Graph Data Base. This package is the data base in which the various graphs are stored. Each graph corresponds to an application package and is not known to the system ahead of time. A graph may be any sequence of nodes and edges

similar to a graph grammar. That is, it may have subgraphs associated with some of its nodes. For those nodes which correspond to user selection points, e.g. menus, the multiple edges leading out of such nodes are labelled. The user input is used to determine which exit edge to follow.

Application Library. This package represents the application packages. It has one entry for each application package operation -- an operation is associated with each user interaction node and is called by the graph handler after receiving input from the user. The system does not know about these operations ahead of time.

4.3 The Fall '80 Semester Course

During the Fall '80 semester the same Junior Level Program Design course is again being taught. The course is following the recommendations outlined in this report. Since the course is still in progress at the time of this writing, a final conclusion is not possible. The basic outline of the course has emerged:

Part I.

In this part of the course, a great deal of emphasis is placed on two foundation stones:

A. Fundamental Programming Techniques. These include data structures and recursive programming. For the most part, this is review material and is covered by the course prerequisites.

B. Formal Specifications. In this part, the emphasis is on predicate calculus. A great deal of time is spent on specifying programs formally and distinguishing between what one does in specifying a program versus what one does in realizing, or implementing, a program to satisfy a specification. This is a relatively new concept for the students and it takes them a while to catch on. It is also fairly difficult for them to get a feeling for a specification language and what it means to be precise in making specifications.

In this part of the course, students are introduced

to the basic ideas of program proofs. They are taught how to take a specification and a proposed realization and then prove that the proposed program does, in fact, implement the specification. While the basic ideas behind program verification are introduced, the course does not attempt to teach the subject in any depth. Program verification is used as a motivation for the need for precise specifications and for well organized implementations.

Program verification does serve as a good bridge. For students who are comfortable with programming, the idea that a specification can be linked to more familiar objects (i.e. programs) helps them understand the difference between specifications and programs and also helps them develop a facility for writing specification.

The basic ideas behind the definition and use of abstract data types in specifications are covered in this part of the course. The talk from section 5 of this report is typical of the material covered in Part I of the course. The book: Jones, C.B., Software Development: a Rigorous Approach, Prentice Hall, 1980 is a reasonable text for this part of the course.

PART II.

In this part of the course, the tone changes dramatically. Part I was quite formal and had a good deal

of notation. It focusses mainly on small programs and their specifications. In this part, the course concentrates on requirements and specification for large systems. Again there are two parts.

A. Requirements. In this part, the idea of user requirements is covered. Requirements are taught as the user's view of what the intended system should do for him. Requirements are stated in user domain language, are about user goals and present the user's world model.

B. Specifications. In this part, the idea of formal specification, as covered in Part I, is applied to large systems. In Part I, only small functions and operations were specificied. Here, systems which are large enough to need requirements are dealt with. The main topics are (1) how to translate requirements which are stated in user domain terms and are about user goals into more formal notation and language, and (2) how to trace from the requirements to the specification to be sure that the goals defined by the requirements will be met by the system described in the specification.

PART III

The earlier parts of the course do not have a great deal of emphasis on Ada. Ada is used as the example programming language in Part I. (But Ada is used more or less as pseudo-code or as a variant of Pascal and no

attempt is made to teach Ada per se.) In Part 2, the notion of Ada packages is introduced on a very informal basis when it is necessary to express external system specifications. But again, Ada as such is not central. In this third part, Ada is much more important. This part of the course is about system architecture, and the Ada package facility is the language used to express architecture. Once again, there are two sub-parts.

A. The Package/Module Concept. Here the package construct is introduced formally and the visible/hidden distinction emphasized. A lot of time is spent discussing the various uses to be made of packages, and it is emphasized that these uses correspond to different sorts of design components. The main design constructs discussed are: encapsulated data types (abstract data types are taught in Part I), encapsulated data objects, abstract machines, libraries, tasks and levels of abstraction. Each of these concepts is discused and its use developed.

B. System Design. In this final section of the course, the design constructs just discussed are joined together with the formal specifications of Part II B to show how large systems can be designed and rigorously justified. In effect, this part of the course shows how large package specifications (i.e. the sort of specification resulting at the top level of an entire system specification) can be broken down and implemented in terms of lower level

packages. This process of "package stepwise refinement" is taught in terms of the package design constructs presented in Part III A of the course.

5. Introduction to Ada Program Design Methodology

The presentation which follows demonstrates a program design strategy which is supported by Ada. It focusses on the concepts of Information Hiding and Abstraction. The presentation shows how the recommended methodology would be carried out on a particular program design problem. It is not the intent of the presentation to focus on the problem itself. The problem is only a vehicle for presenting the design strategy.

ABSTRACT DATA TYPES

&

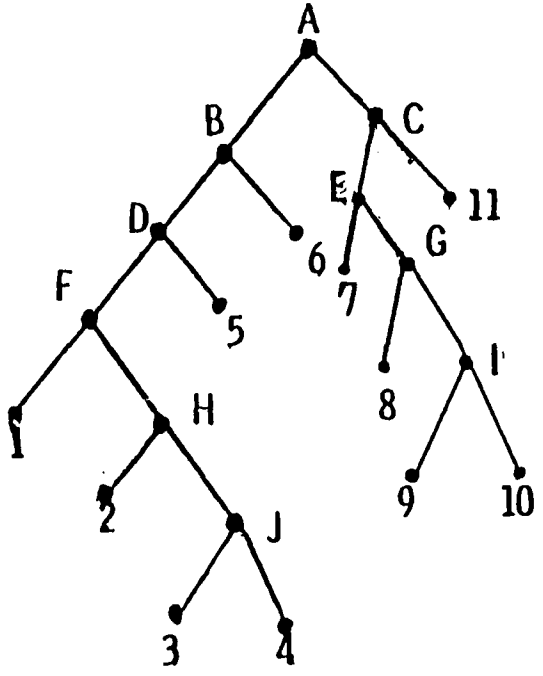
INFORMATION HIDING

THIS TALK PRESENTS A PROGRAM DESIGN METHODOLOGY. THE METHODOLOGY IS BASED ON THE CONCEPTS OF ABSTRACT DATA TYPES, INFORMATION HIDING AND LEVELS OF ABSTRACTION.

THE METHODOLOGY WILL BE PRESENTED BY DESIGNING A PROGRAM FOR A SAMPLE PROBLEM. THE SAMPLE PROBLEM SELECTED IS RELATIVELY SIMPLE. THE PURPOSE OF THE TALK IS NOT TO SOLVE THIS PARTICULAR PROBLEM BUT TO DEMONSTRATE THE METHODOLOGY. THE METHODOLOGY HAS ITS GREATEST PAYOFFS ON VERY LARGE PROBLEMS.

PROBLEM

GIVEN A BINARY TREE, COUNT ITS LEAVES.

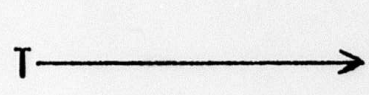


THIS BINARY TREE HAS 11 LEAVES, I.E. TERMINAL "NODES". THERE ARE ALSO INTERNAL "NODES" (A-J) WHICH ARE NOT LEAVES.

AS PART OF THE SPECIFICATION OF THE PROBLEM, WE ARE GIVEN

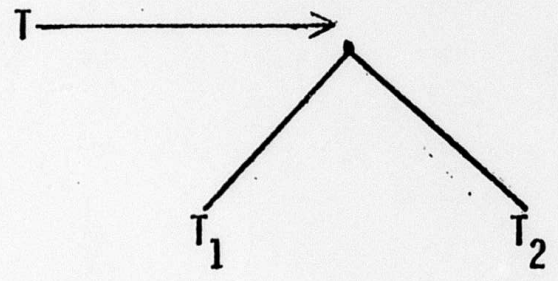
1. WHAT IS A TREE (I.E. THE INPUT)

   A TREE T IS EITHER A LEAF (ONLY)



OR

A TREE CONSISTS OF TWO SUBTREES



(T_1 AND T_2 ARE THE SUBTREES OF T)

2. THE NUMBER OF LEAVES ON A TREE T IS DEFINED ACCORDING TO THESE TWO CASES

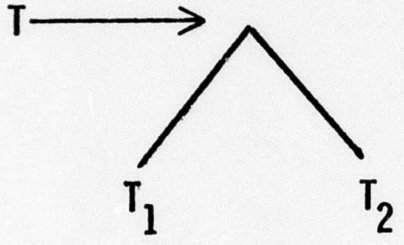
LEAVES (T) = 1,

IF T IS A LEAF

T ⟶ HAS ONE LEAF, ITSELF

LEAVES (T) = LEAVES (T_1) + LEAVES (T_2),

IF T HAS T_1 AND T_2 AS SUBTREES



T HAS AS MANY LEAVES
AS T_1 AND T_2 TOGETHER

THIS WOULD BE A TRIVIAL PROBLEM TO CODE IN A LANGUAGE WITH RECURSION. EVEN WITHOUT RECURSION THIS IS NOT A DIFFICULT PROBLEM. WE ARE TAKING IT HERE ONLY AS A VEHICLE TO PRESENT THE CONCEPTS OF ABSTRACT DATA TYPES AND INFORMATION HIDING.

TO ILLUSTRATE THE PROGRAM DESIGN MEHTODOLOGY PROPOSED, WE WILL DEVELOP A DESIGN FOR THE SOLUTION OF THIS PROBLEM WITHIN THE BOUNDS OF THE FOLLOWING (FAMILIAR) CONSTRAINTS.

PROGRAM DESIGN CONSTRAINTS

- THE PROGRAM SHALL BE ORGANIZED INTO "LEVELS" WHERE EACH LEVEL IS COMPLETE IN ITSELF.

- THE SOLUTION TO THE PROBLEM SHALL BE EXPRESSED IN TERMS OF TOP LEVEL CONSTRUCTS

- IT SHALL BE POSSIBLE TO

  DESIGN
  REVIEW
  VERIFY
  CODE
  TEST

  EACH LEVEL INDEPENDENT OF OTHER LEVELS.

- THE DESIGN AND IMPLEMENTATION OF LOWER LEVELS SHALL NOT AFFECT THE DESIGN CORRECTNESS OF HIGHER LEVELS.

TO ACHIEVE THESE DESIGN GOALS WE SHALL DO OUR DESIGN WITHIN THE FOLLOWING ADDITIONAL CONSTRAINTS. WE WILL USE A DESIGN LANGUAGE WITH THESE CAPABILITIES.

- THE DESIGN LANGUAGE HAS THE STANDARD SINGLE-ENTRY/SINGLE-EXIT CONTROL STRUCTURES:

  A) SEQUENCE B) SELECTION (IF/CASE) C) ITERATION (LOOP, WHILE, ...)

- THE DESIGN LANGUAGE DOES NOT SUPPORT RECURSION.

- THE DESIGN LANGUAGE HAS NO PRE-DEFINED OPERATIONS (NOT EVEN ASSIGNMENT: A := B)

- THE DESIGN LANGUAGE HAS NO PRE-DEFINED OBJECTS.

- THE DESIGN LANGUAGE HAS FACILITIES THROUGH WHICH WE CAN DEFINE OPERATIONS AND OBJECTS AS NEEDED.

SINCE OUR DESIGN LANGUAGE PROVIDES US WITH NEITHER PRE-DEFINED OBJECTS NOR OPERATIONS, IS OUR TASK TO SPECIFY THE OBJECTS AND OPERATIONS WE WANT TO USE TO DO THE JOB.

WE CANNOT BEGIN BY SAYING "LETS PICK A PROGRAMMING LANGUAGE." THE PROBLEM IS NOT SUPPOSED TO BE A PROGRAMMING PROBLEM BUT A PROGRAM DESIGN PROBLEM. THE SOLUTION WE COME UP WITH SHOULD BE ABLE TO BE IMPLEMENTED IN VARIOUS PROGRAMMING LANGUAGES; IT SHOULD NOT BE BASED ON THE SPECIFIC FEATURES OF ONE LANGUAGE.

IN OTHER WORDS, THE ONLY PRE-DEFINED TOOLS WE HAVE ARE CONTROL STRUCTURES SUCH AS IF, CASE, WHILE, LOOP, ETC. WE CANNOT ASSUME WE HAVE OBJECTS OR OPERATIONS GIVEN TO US.

THE PURPOSE OF THESE CONSTRAINTS IS TO MAKE ALL OUR DESIGN CONSTRUCTS EXPLICIT AND VISIBLE.

HOW SHOULD ONE APPROACH A PROGRAM DESIGN PROBLEM?

1. DEVELOP AN INFORMAL STRATEGY
2. EXPRESS THAT STRATEGY IN SOME FORMAL NOTATION
3. VERIFY THE DESIGN

(VERY) INFORMAL STRATEGY FOR COUNTING TREE LEAVES

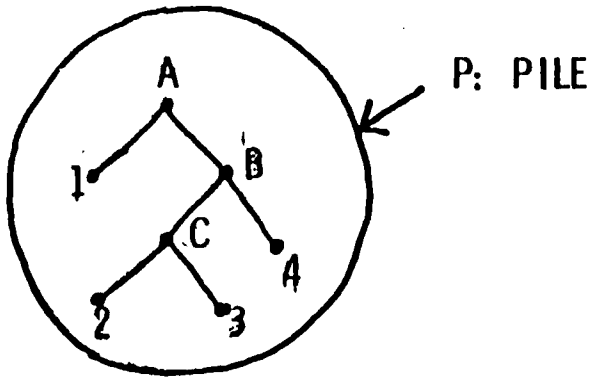
A) WE WILL KEEP A "PILE" OF THESE PARTS OF THE TREE WHICH HAVE NOT YET BEEN COUNTED. INITIALLY THE PILE CONSISTS OF JUST THE ORIGINAL TREE.

B) WE WILL REPEATEDLY PULL THINGS OFF THE PILE. AS WE PULL SOMETHING OFF THE PILE, WE WILL LOOK AT IT. IF IT IS ITSELF A TREE (AND NOT A LEAF), WE WILL BREAK IT IN HALF AND PUT BOTH HALVES BACK ONTO THE PILE. IF IT IS A LEAF. WE WILL COUNT IT (AND THROW IT AWAY).

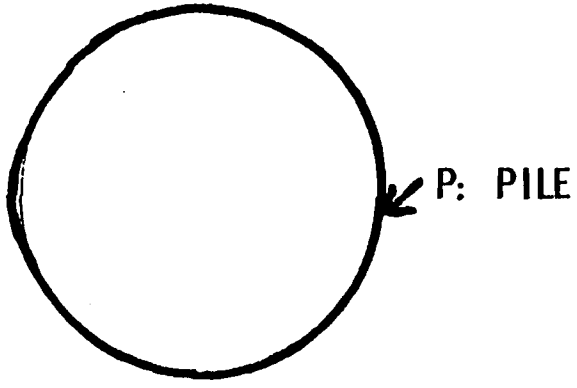
C) WHEN THE PILE IS EMPTY, WE ARE DONE.

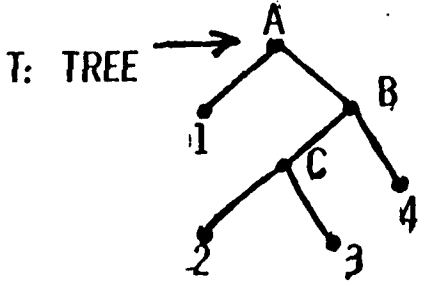
EXAMPLE

1. TO: ORIGINAL TREE



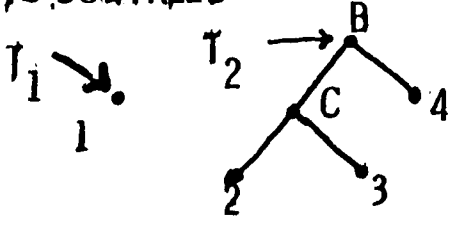
2. PULL SOMETHING OUT OF THE PILE AND LOOK AT IT. THE ONLY THING IN THE PILE IS THE ORIGINAL TREE, SO WE TAKE IT.

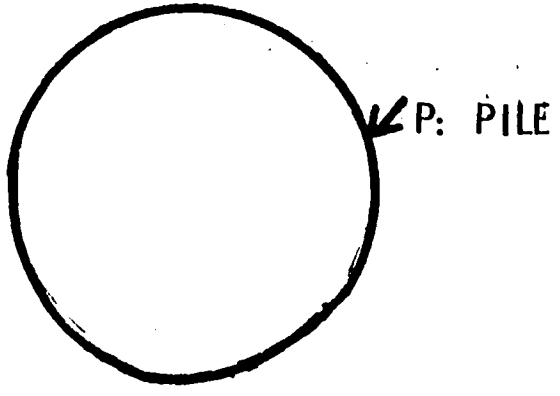




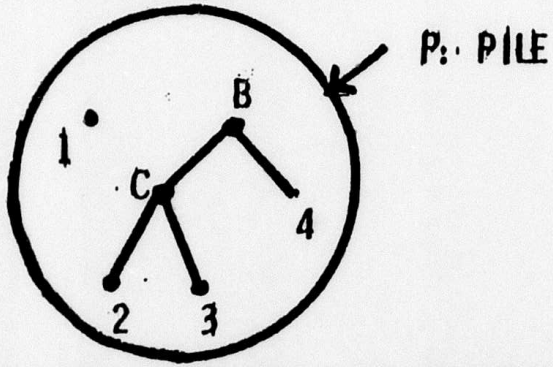
3. SINCE IT IS A TREE (WITH SUBTREES AND NOT JUST A SINGLE LEAF), BREAK IT INTO ITS SUBTREES

TWO SUBTREES

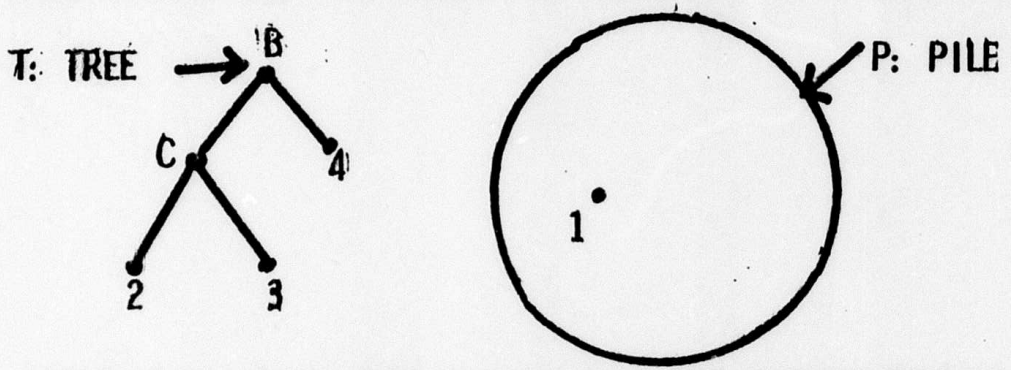




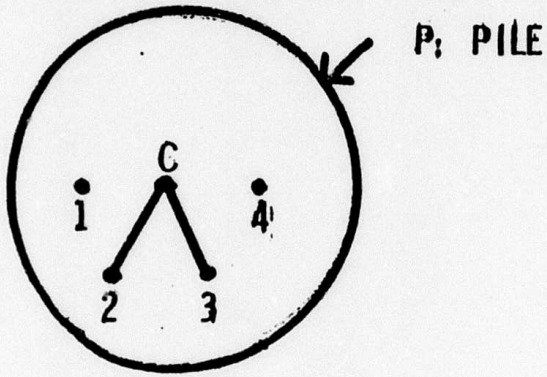
4. PUT THOSE TWO SUBTREES BACK ONTO THE PILE



5. TAKE SOMETHING FROM THE PILE. (LETS SAY WE TAKE THE TREE B).

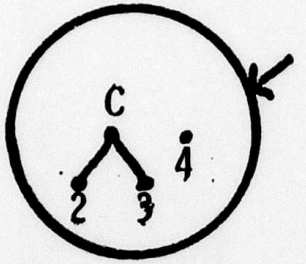


6. SINCE IT IS A TREE, BREAK IT IN HALF AND PUT THE TWO HALVES BACK ONTO THE PILE



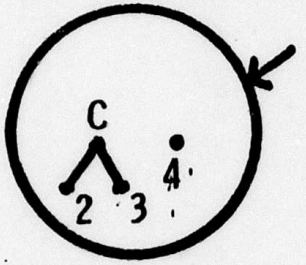
7. TAKE SOMETHING FROM THE PILE. (LET'S SAY WE TAKE THE LEAF 1 THIS TIME.)

T: TREE → 1



P: PILE

8. SINCE IT IS A LEAF, INCREMENT THE COUNTER AND THROW THE LEAF AWAY



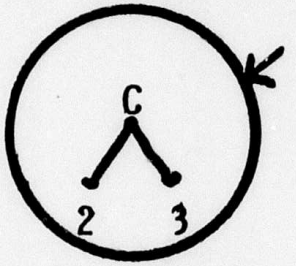
P: PILE

C: COUNTER

1

9. TAKE SOMETHING FROM THE PILE. (LET'S SAY WE TAKE THE LEAF 4).

T: TREE → 4

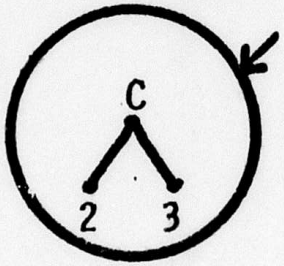


P: PILE

1

C: COUNTER

10. SINCE IT IS A LEAF, INCREMENT THE COUNTER AND THROW IT AWAY.

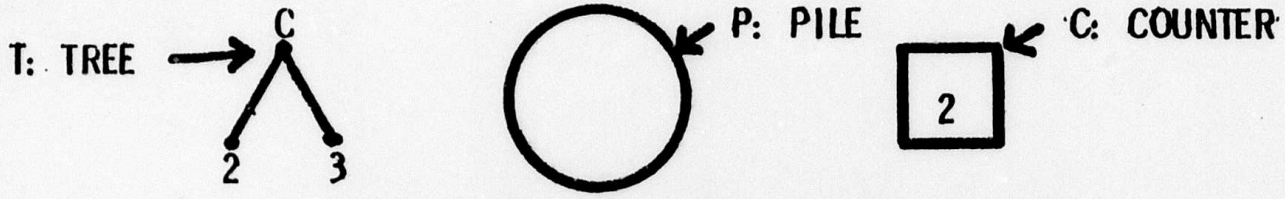


P: PILE

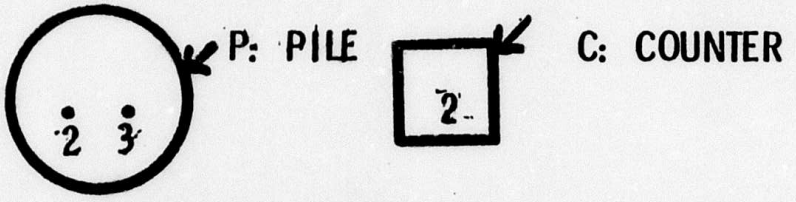
C: COUNTER

2

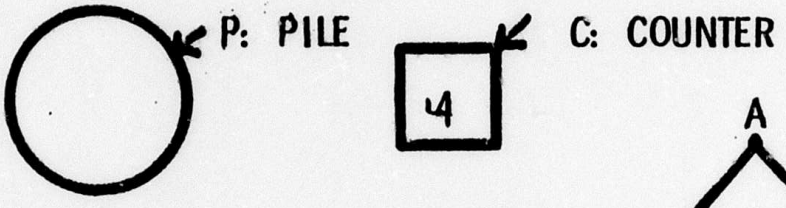
11. TAKE SOMETHING FROM THE PILE. (THIS TIME THE ONLY THING LEFT IN THE PILE IS THE TREE C.)



12. SINCE IT IS A TREE, BREAK IT IN HALF AND PUT THE TWO HALVES BACK ONTO THE PILE

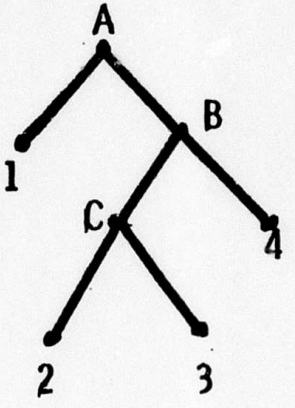


13. & 14. REPEAT THE PROCESS WITH THE TWO REMAINING LEAVES UNTIL THEY ARE BOTH COUNTED.



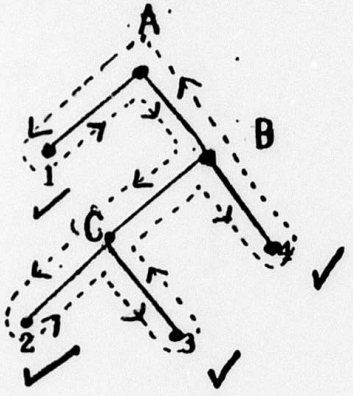
15. FINISHED.

THE ORIGINAL TREE DID HAVE EXACTLY FOUR LEAVES.



ARE OTHER STRATEGIES POSSIBLE? YES.

2. "TRAVERSE" THE TREE -- I.E. WALK THROUGH THE ENTIRE TREE IN SOME ORDER AND COUNT THE LEAVES



3. BREAK THE TREE UP INTO "NODES," WHERE A NODE IS EITHER AN INTERNAL NODE OR A LEAF. LOOK AT ALL THE NODES AND COUNT THE ONES WHICH ARE LEAVES

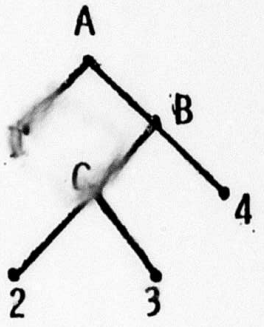
A 1 B C 4 2 3

✓ ✓ ✓ ✓

WHAT ABOUT THE VERSION OF 3 WHICH SAYS:

3' REPRESENT THE TREE BY BUILDING AN ARRAY OF ITS NODES. SCAN DOWN THE ARRAY AND COUNT THE LINES WITH *'S.

TREE



REPRESENTATION

	LEFT	RIGHT	
A	1	B	
1	*	*	(A LEAF)
B	C	4	
C	2	3	
4	*	*	(A LEAF)
2	*	*	(A LEAF)
3	*	*	(A LEAF)

SUCH AN APPROACH IS NOT ACCEPTABLE. IT VIOLATES OUR CONSTRAINTS

"THE SOLUTION TO THE PROBLEM SHALL BE EXPRESSED IN TERMS OF TOP LEVEL CONSTRUCTS"

"THE DESIGN AND IMPLEMENTATION OF LOWER LEVELS SHALL NOT AFFECT THE DESIGN CORRECTNESS OF HIGHER LEVELS."

THE PROBLEM IS THAT: 1) THE TOP LEVEL OF THE PROBLEM, THE TERMS IN WHICH THE PROBLEM IS GIVEN, DEALS WITH TREES AND LEAVES

2) THE PROPOSED SOLUTION IS STATED IN TERMS OF ARRAYS AND *'S

SO 1) WE HAVE NOT EXPRESSED THE SOLUTION TO THE PROBLEM IN ITS OWN TERMS.

2) WE ARE VULNERABLE TO A CHANGE IN OUR LOWER LEVEL DECISIONS. SHOULD WE CHANGE OUR REPRESENTATION OF TREES FROM ARRAYS TO SOMETHING ELSE, THE SOLUTION NO LONGER WORKS. FOR EXAMPLE, TREES CAN ALSO BE REPRESENTED AS LINKED LISTS.

INFORMATION HIDING IS THE DESIGN DISCIPLINE WHICH HIDES INFORMATION DEVELOPED ON ONE LEVEL (SUCH AS THE REPRESENTATION OF TREES BY ARRAYS) FROM ITS USE ON ANOTHER LEVEL.

IT IS A POOR DESIGN PRACTICE TO MAKE A HIGHER LEVEL DESIGN STRATEGY DEPENDENT ON A LOWER LEVEL IMPLEMENTATION DECISION. IF WE FOLLOWED STRATEGY 3' OUR TOP LEVEL DESIGN DECISION WOULD BE TOTALLY VULNERABLE TO MINOR CHANGES IN OUR LOWER LEVEL IMPLEMENTATION DECISION ABOUT HOW TO REPRESENT TREES.

INFORMATION HIDING IS THE PRACTICE OF KEEPING LOWER LEVEL DECISIONS INVISIBLE SO THAT HIGHER LEVEL STRATEGIES ARE NOT BASED UPON THEM.

HOW SHOULD ONE APPROACH A PROGRAM DESIGN PROBLEM?

1. DEVELOP AN INFORMAL STRATEGY
2. EXPRESS THAT STRATEGY IN SOME FORMAL NOTATION
3. VERIFY THE DESIGN

2. FORMALIZE THE STRATEGY. (RECALL OUR ORIGINAL STRATEGY WITH THE PILE)

WHAT IS NEEDED TO FORMALIZE THE SELECTED STRATEGY?

1. DEFINE THE OBJECTS BEING MANIPULATED
2. DEFINE THE OPERATIONS TO BE PERFORMED ON THOSE OBJECTS
3. DEFINE THE CONTROL ORGANIZATION IN WHICH THOSE OPERATIONS ARE EMBEDDED TO CARRY OUT THE STRATEGY.

1. WHAT ARE THE OBJECTS BEING MANIPULATED?

(VERY) INFORMAL STRATEGY FOR COUNTING TREE LEAVES

A) WE WILL KEEP A "PILE" OF THESE PARTS OF THE TREE WHICH HAVE NOT YET BEEN COUNTED. INITIALLY THE PILE CONSISTS OF JUST THE ORIGINAL TREE.

B) WE WILL REPEATEDLY PULL THINGS OFF THE PILE. AS WE PULL SOMETHING OFF THE PILE, WE WILL LOOK AT IT. IF IT IS ITSELF A TREE (AND NOT A LEAF), WE WILL BREAK IT IN HALF AND PUT BOTH HALVES BACK ONTO THE PILE. IF IT IS A LEAF, WE WILL COUNT IT (AND THROW IT AWAY).

C) WHEN THE PILE IS EMPTY, WE ARE DONE.

OBJECTS

A) TREES (INCLUDING SUBTREES AND LEAVES)

B) PILE

C) COUNTER (NOT EXPLICITLY MENTIONED)

THERE ARE A TOTAL 6 OBJECTS

TREES: ORIGINAL TREE

INSPECTED TREE

LEFT SUB-TREE

RIGHT SUB-TREE

PILES: THE ONE PILE

COUNTER: THE ONE COUNTER

2. WHAT ARE THE OPERATIONS BEING PERFORMED?

A) WE WILL KEEP A "PILE" OF THESE PARTS OF THE TREE WHICH HAVE NOT YET BEEN COUNTED. INITIALLY THE PILE CONSISTS OF JUST THE ORIGINAL TREE.

B) WE WILL REPEATEDLY PULL THINGS OFF THE PILE. AS WE PULL SOMETHING OFF THE PILE, WE WILL LOOK AT IT. IF IT IS ITSELF A TREE (AND NOT A LEAF), WE WILL BREAK IT IN HALF AND PUT BOTH HALVES BACK ONTO THE PILE. IF IT IS A LEAF, WE WILL COUNT IT (AND THROW IT AWAY).

C) WHEN THE PILE IS EMPTY, WE ARE DONE.

OPERATIONS

A) FOR TREES:

I) FOR ARBITRARY TREE/LEAF SELECTED FROM THE PILE, IS IT A TREE OR A LEAF

II) FOR TREES, TEAR IT IN HALF;

III) FOR LEAVES, THROW IT AWAY;

B) FOR PILE:

I) IS IT EMPTY;

II) PULL SOMETHING OFF IT;

III) PUT SOMETHING ONTO IT;

C) FOR COUNTER:

I) INCREMENT IT (BY ONE).

SUMMARIZING BOTH OBJECTS AND OPERATIONS:

TYPE TREE

OPERATIONS	IS-LEAF (T)	--	(TAKES A TREE (OR LEAF) AND SAYS YES OR NO, I.E. TRUE OR FALSE)
	TEAR-IN-HALF (T, L, R)	--	(TAKES A TREE T AND TEARS IT IN HALF. ITS LEFT HALF IS GIVEN TO L; ITS RIGHT HALF IS GIVEN TO R)
	DISCARD (L)	--	(TAKES A LEAF AND THROWS IT AWAY)

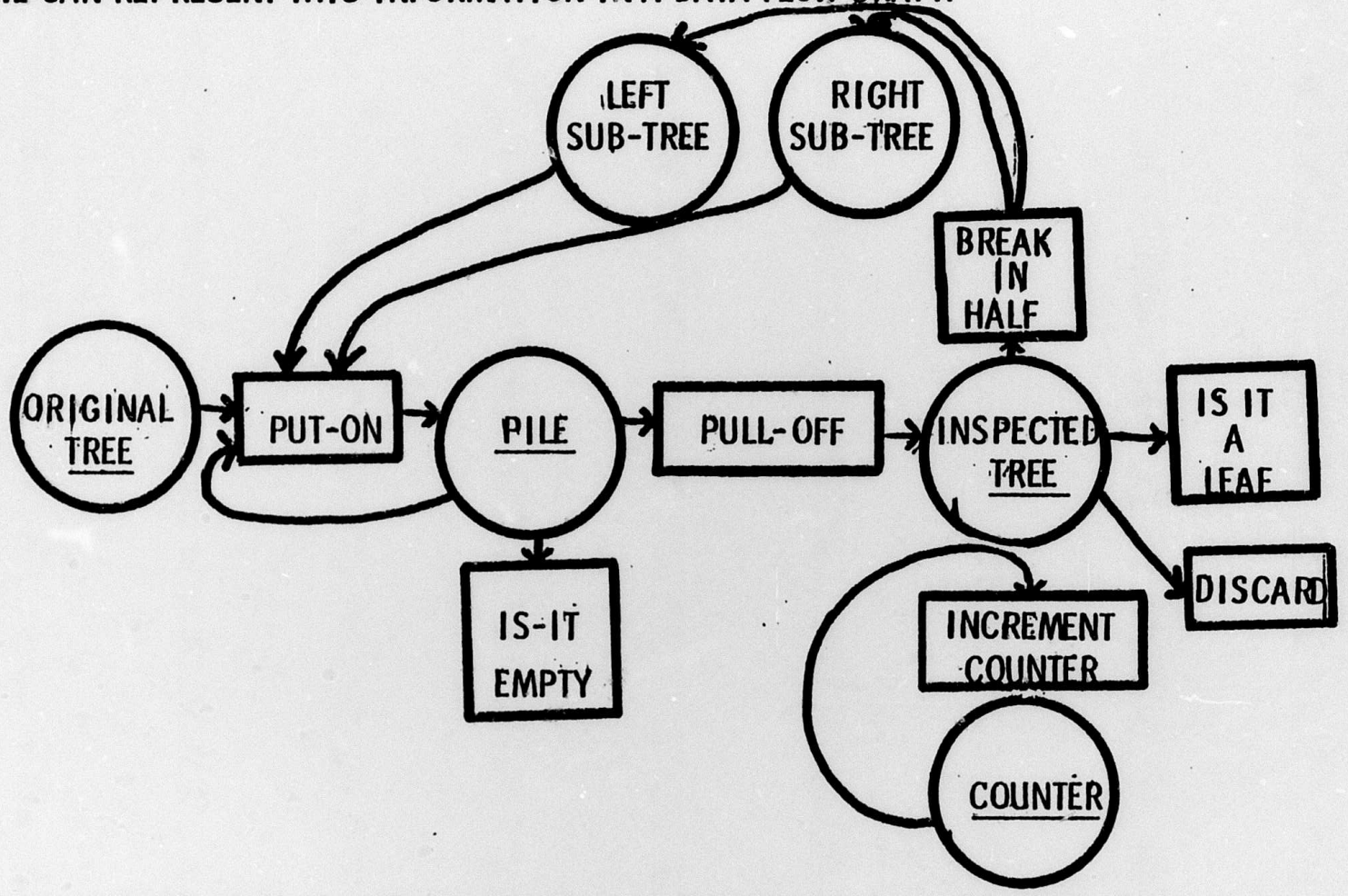
TYPE PILE

OPERATIONS	IS-EMPTY (P)	--	(TAKES A PILE AND DETERMINES IF IT IS EMPTY, I.E. TRUE OR FALSE)
	PUT-ON (T, P)	--	(TAKES A TREE T AND A PILE P AND THROWS T ONTO P)
	PULL-OFF (T, P)	--	(TAKES A PILE P; REMOVES ONE OF ITS TREES AND GIVES IT TO T)

TYPE COUNTER

	INCR(C)	--	(TAKES A COUNTER C AND ADDS 1 TO IT)

WE CAN REPRESENT THIS INFORMATION IN A DATA FLOW GRAPH



THIS GRAPH DOES NOT SHOW A SEQUENCE OF EVENTS OR ANY TYPE OF CONTROL ORGANIZATION.

ALL IT SHOWS ARE 1) THE OBJECTS; 2) THE OPERATIONS; AND 3) WHICH OPERATIONS ARE (EVER POSSIBLY) APPLIED TO WHICH OBJECTS.

THE CONTROL FLOW

```
P: PILE;
TO, T, L, R: TREE;
C: COUNTER;



PUT-ON (TO, P);
LOOP

   EXIT WHEN IS-EMPTY (P);
   PULL-OFF (T, P)
   IF IS-LEAF (T) THEN INCR (C); DISCARD (T);
      ELSE TEAR-IN-HALF (T, L, R ); PUT-ON (L, P); PUT-ON (R, P);
   END IF;

END LOOP;
```

A TOP DOWN, STRUCTURED PROGRAMMING DESIGN METHODOLOGY MIGHT HAVE GOT US TO THIS SAME POINT. USING TOP DOWN DESIGN ONE WOULD NOW PROCEED TO DEFINE THE PROGRAMS WHICH PERFORM THE OPERATIONS INDICATED. ONE MIGHT DECIDE ON A REPRESENTATION STRATEGY FOR TREES AND PILES (AS IN STRATEGY 3') AND PROCEED TO IMPLEMENT THE OPERATIONS IN TERMS OF THOSE REPRESENTATIONS.

THE APPROACH PRESENTED HERE DOES NOT PURSUE FURTHER IMPLEMENTATION AT THIS POINT. INSTEAD WE STEP BACK AND ASK WHAT IS NEEDED TO SHOW THAT THE DESIGN PRODUCED SO FAR IS CORRECT.

HOW SHOULD ONE APPROACH A PROGRAM DESIGN PROBLEM?

1. DEVELOP AN INFORMAL STRATEGY
2. EXPRESS THAT STRATEGY IN SOME FORMAL NOTATION
3. VERIFY THE DESIGN

3. HOW CAN WE VERIFY THE DESIGN? WHAT IS NEEDED?

WE NEED SOMETHING WHICH TELLS US WHAT THE EFFECTS ARE OF THE OPERATIONS WE HAVE DEFINED AND WHAT RELATES THEM TO EACH OTHER. ESPECIALLY WE WANT TO KNOW WHAT THEY HAVE TO DO WITH THE NUMBER OF LEAVES AT VARIOUS PLACES.

WE WANT TO SPECIFY RELATIONSHIPS AMONG THE VARIOUS OPERATIONS AND OBJECTS.

NOTATION

IN ORDER TO EXPRESS SPECIFICATIONS OF OPERATIONS, WE NEED NOTATION TO TALK ABOUT THE AFFECT OPERATIONS HAVE WHEN PERFORMED IN A PROGRAM. WE WILL USE {..} TO EMBED IN A PROGRAM CLAIMS MADE ABOUT THE VALUES OF VARIABLES.

EXAMPLE

```
A:=B;
B:=B - 1;
{A > B}
```

THE LINE {A > B} IS CLAIMING THAT AT THE POINT INDICATED A IS GREATER THAN B. (AND, OF COURSE, THE CLAIM WILL BE TRUE.)

THE MAIN THING WE ARE CONCERNED ABOUT HERE IS THE NUMBER OF LEAVES AT VARIOUS PLACES.

LET'S DEFINE THE FUNCTION "LEAVES" WHICH "KNOWS" THAT INFORMATION. WHAT DO WE KNOW ABOUT "LEAVES"?

1. LEAVES OF TREES.

FROM OUR ORIGINAL PROBLEM SPECIFICATION WE KNOW

IF IS-LEAF (T) THEN {LEAVES (T) = 1}
ELSE {LEAVES (T) = X}
TEAR-IN-HALF (T, L, R)
{LEAVES (L) + LEAVES (R) = X}

THIS STATEMENT SPECIFIES RELATIONSHIPS BETWEEN:

A) THE OPERATIONS "IS-LEAF" AND "TEAR-IN-HALF"; AND

B) THE TREE ATTRIBUTE "LEAVES".

THESE RELATIONSHIPS ARE OFTEN CALLED "AXIOMS."
THEY DEFINE THE MEANING OF IS-LEAF AND TEAR-IN-HALF IN TERMS OF THE ATTRIBUTE LEAVES.

2. LEAVES OF PILES. LET P BE A PILE, T A TREE.

A) DEFINE LEAVES (P): $\sum_{T_i \in P}$ LEAVES (T_i)

I.E. THE NUMBER OF LEAVES IN A PILE IS DEFINED TO BE THE SUM OF THE LEAVES OF THE TREES IN THE PILE.

B) {LEAVES(T) = X & LEAVES(P) = Y }
PUT-ON (T, P)
{LEAVES (P) = X + Y}

C) IF IS-EMPTY·(P) THEN {LEAVES (P) = 0}
ELSE {LEAVES (P) = X}
PULL-OFF (T, P)
{LEAVES (T) + LEAVES (P) = X}

3. INFORMATION ABOUT COUNTERS.

LET C BE A COUNTER

```
{VALUE (C) = X}
 INCR (C);
{VALUE (C) = X+1}
```

ABSTRACT DATA TYPES

WITH THESE AXIOMS, WE HAVE COMPLETED OUR DEFINITIONS OF TREES, PILES AND COUNTERS AS DATA TYPES. THESE DEFINITIONS DEFINE TREES, PILES AND COUNTERS AS ABSTRACT DATA TYPES.

AN ABSTRACT DATA TYPE IS CHARACTERIZED (AS WE HAVE DONE) BY:

1) A COLLECTION OF OPERATIONS;
2) A COLLECTION OF AXIOMS WHICH FORMALIZE THE EFFECTS OF THE OPERATIONS IN TERMS OF OTHER, KNOWN CONCEPTS.

FOR TREES, FOR EXAMPLE: THE OPERATIONS ARE: "IS-LEAF" AND "TEAR-IN-HALF"
THE AXIOM IS THE STATEMENT WHICH STATES HOW THE ATTRIBUTE "LEAVES" IS RELATED TO THESE OPERATIONS

```
    P: PILE;
    TO, T, L, R: TREE;
    C: COUNTER;
→   {X=LEAVES (TO)}


    PUT-ON (TO, P);
    LOOP
→     {X=VALUE (C) + LEAVES (P)}
      EXIT WHEN IS-EMPTY (P);
      PULL-OFF (T, P)
      IF IS-LEAF (T) THEN INCR (C); DISCARD (T);
        ELSE TEAR-IN-HALF (T, L, R ); PUT-ON (L, P); PUT-ON (R, P);
      END IF;

    END LOOP;

→   {X=VALUE (C)}
```

TO VERIFY OUR PROGRAM WE WOULD LIKE TO DEMONSTRATE THE THREE LINES SHOWN. MAINLY, WE WANT TO SHOW THAT IF X IS THE NUMBER OF LEAVES ON THE ORIGINAL TREE, X WILL BE THE FINAL VALUE IN THE COUNTER AT THE END OF THE PROGRAM.

```
P: PILE;
TO, T, L, R: TREE;
C: COUNTER;
{X=LEAVES (TO)}

{LEAVES (P)=0;  VALUE (C)=0}
PUT-ON (TO, P);
LOOP
   {X=VALUE (C) + LEAVES (P)}
   EXIT WHEN IS-EMPTY (P);
   PULL-OFF (T, P)
   IF IS-LEAF (T) THEN INCR (C); DISCARD (T);
      ELSE TEAR-IN-HALF (T, L, R ); PUT-ON (L, P); PUT-ON (R, P);
   END IF;

END LOOP;
{X=VALUE (C) + LEAVES (P) & LEAVES (P) = 0}
{X=VALUE (C)}
```

AT THIS POINT WE HAVE FOUND A POTENTIAL BUG IN THE PROGRAM. WE NEVER RESET THE COUNTER OR THE PILE!

TO FIX IT WE NEED TWO MORE OPERATIONS.

A) RESET (P: PILE)

{LEAVES (P) = 0}

B) RESET (C: COUNTER)

{LEAVES (C) = 0}

```
P: PILE;
TO, T, L, R: TREE;
C: COUNTER;
{X=LEAVES (TO)}
RESET (C); RESET (P);
{LEAVES (P)=0;  VALUE (C)=0}
PUT-ON (TO, P);
{X=LEAVES (P);  VALUE(C)=0}
LOOP
   {X=VALUE (C) + LEAVES (P)}
   EXIT WHEN IS-EMPTY (P);
   PULL-OFF (T, P)
   IF IS-LEAF (T) THEN INCR (C); DISCARD (T);
      ELSE TEAR-IN-HALF (T, L, R ); PUT-ON (L, P); PUT-ON (R, P);
   END IF;
   {X=VALUE (C) + LEAVES (P)}
END LOOP;
{X=VALUE (C) + LEAVES (P);  LEAVES (P) = 0}
{X=VALUE (C)}
```

THIS CONCLUDES THE VERIFICATION OF THE PROGRAM.

THE SIGNIFICANCE OF THIS VERIFICATION IS TWO FOLD.

1) THE PROGRAM HAS BEEN VERIFIED IN TERMS OF THE ABSTRACT DEFINITIONS OF THE DATA TYPES AND OPERATIONS. NO IMPLEMENTATION DETAILS HAD TO BE CONSIDERED.

2) THE VERIFICATION PROCESS POINTED OUT WHAT CONSTRAINTS WE NEED ON THE OPERATIONS. THAT IS, THE OPERATIONS MUST CONFORM TO THE AXIOMS. IN EFFECT, THESE AXIOMS BECOME THE REQUIREMENTS SPECIFICATIONS FOR THE OPERATIONS.

IN OUTLINE, THE ENTIRE PROGRAM APPEARS:

SPECIFICATION FOR TREE IS

OPERATIONS

AXIOMS

SPECIFICATION FOR PILE IS

OPERATIONS

AXIOMS

SPECIFICATION FOR COUNTER IS

OPERATIONS

AXIOMS

DECLARATIONS P: PILE; TO, T, L, R: TREE; C: COUNTER;

{INPUT CONDITIONS}

OPERATIONAL PROGRAM

{OUTPUT CLAIMS}

WHAT HAVE WE DONE?

1) WE HAVE DEFINED A LEVEL OF ABSTRACTION CONSISTING OF THE SPECIFICATIONS FOR TREE, PILE AND COUNTER.

2) WE WROTE A PROGRAM ENTIRELY ON THAT LEVEL OF ABSTRACTION.

3) WE VERIFIED THE PROGRAM STRICTLY IN TERMS OF THE SPECIFICATIONS FOR THAT LEVEL OF ABSTRACTION.

AT NO TIME DID WE TALK ABOUT, NOR NEED TO KNOW, HOW ANY OF THE OBJECTS OR OPERATIONS WERE IMPLEMENTED. ALL WE NEEDED WAS THEIR FUNCTIONAL SPECIFICATIONS, I.E. THE AXIOMS.

PROGRAM DESIGN CONSTRAINTS

- THE PROGRAM SHALL BE ORGANIZED INTO "LEVELS" WHERE EACH LEVEL IS COMPLETE IN ITSELF.

- THE SOLUTION TO THE PROBLEM SHALL BE EXPRESSED IN TERMS OF TOP LEVEL CONSTRUCTS

- IT SHALL BE POSSIBLE TO

  DESIGN
  REVIEW
  VERIFY
  CODE
  TEST

  EACH LEVEL INDEPENDENT OF OTHER LEVELS.

- THE DESIGN AND IMPLEMENTATION OF LOWER LEVELS SHALL NOT AFFECT THE DESIGN CORRECTNESS OF HIGHER LEVELS.

HAVE WE SATISFIED OUR DESIGN STANDARDS?

1) THE PROGRAM IS ORGANIZED INTO LEVELS.

WHAT WE HAVE BEEN DISCUSSING IS THE TOP LEVEL. OTHER LEVELS ARE REQUIRED TO SUPPORT IT. AT SOME POINT, PROGRAMS MUST BE WRITTEN WHICH ACTUALLY PERFORM THE OPERATIONS WE HAVE SPECIFIED. THOSE PROGRAMS WILL NEED TO WORK IN TERMS OF SOME REPRESENTATION OF THE OBJECTS MANIPULATED.

THAT IS, WE MAY CHOOSE TO REPRESENT A COUNTER AS A STRING OF 1'S: '1111'=4. IN THAT CASE THE OPERATION INCR(C) CONCATENATES AN ADDITIONAL '1' TO THE END OF C. THIS IMPLEMENTATION LEVEL IS A SECOND LEVEL.

THE TOP LEVEL IS COMPLETE IN ITSELF. THE MEANINGS OF THE OPERATIONS ARE ALL DEFINED IN TERMS OF EACH OTHER ON THAT LEVEL.

2) THE SOLUTION TO THE PROBLEM IS EXPRESSED IN TERMS OF TOP LEVEL CONSTRUCTS.

THAT CONSTRAINT IS SATISFIED. THE TOP LEVEL IS REALLY JUST A FORMALIZATION OF THE CONCEPTS GIVEN IN THE PROBLEM DEFINITION, I.E. LEAVES AND TREES. WE ADDED A FEW ADDITIONAL CONCEPTS WHICH WE RELATED FORMALLY TO THE GIVEN ONES. THEN WE EXPRESSED THE PROBLEM SOLUTION USING THESE TOP LEVEL CONCEPTS.

3) IT IS POSSIBLE TO: DESIGN, REVIEW, VERIFY, CODE AND TEST EACH LEVEL SEPARATELY.

EACH LEVEL HAS A FORMAL SPECIFICATION. IT IS THAT SPECIFICATION WHICH PERMITS EACH LEVEL TO BE TREATED INDEPENDENTLY OF THE OTHERS.

EACH LEVEL MAY BE DESIGNED, ETC. WITH RESPECT TO ITS OWN SPECIFICATION AND WITHOUT REGARD TO THE OTHER LEVELS.

4) THE DESIGN AND IMPLEMENTATION CHOICES MADE FOR THE LOWER LEVELS WILL NOT AFFECT THE CORRECTNESS OF HIGHER LEVELS.

IT DOES NOT MATTER HOW WE REPRESENT THE COUNTER AS LONG AS IT MEETS ITS SPECIFICATION. WE COULD REPRESENT IT AS A STRING OF 1'S, AS SUGGESTED. MORE LIKELY WE COULD REPRESENT IT AS AN INTEGER. WE COULD REPRESENT IT AS THE POSITION OF THE LEFT MOST 1 BIT IN A WORD. [IN THIS CASE INCR(C) WOULD BE SHIFT (C).] THE POINT IS, IT DOESN'T MATTER. THE HIGHER LEVEL IS INDEPENDENT OF ANY SUCH DECSION.

CONCLUSION

INFORMATION HIDING, ABSTRACT DATA TYPES AND LEVELS OF ABSTRACTION PROVIDE METHODOLOGICAL CONCEPTS AND TOOLS TO SIMPLIFY AND RATIONALIZE THE SYSTEM SPECIFICATION, DESIGN AND MANAGEMENT PROCESS.