5-1-1985

Programming language as a tool for software design

Thavisakdi Petcharawises
Atlanta University

PROGRAMMING LANGUAGE AS A TOOL FOR SOFTWARE DESIGN

A THESIS

SUBMITTED TO THE FACULTY OF ATLANTA UNIVERSITY

FOR PARTIAL FULFILLMENT OF THE REQUIREMENT FOR

THE DEGREE OF MASTER OF SCIENCE

BY

THAVISAKDI PETCHARAWISES

DEPARTMENT OF MATHEMATICAL SCIENCE

ATLANTA UNIVERSITY

ATLANTA, GEORGIA

MAY 1985

ACKNOWLEDGEMENT

I would like to express gratitude to my advisor Dr. Nazir A. Warsi for his valuable help in completion of this work. I would also like to thank Drs. Benjamin Martin, C. B. Setzer, Robert Bozeman, and Mr. John Kao for their guidance and encouragement.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The production of a large program involves writing several thousand lines of code. Consequently, it adds a new level of complexity and challenge as compared to the development of smaller programs. The techniques which work well with the small programs do not apply to the larger programs. The approaches to the large software production have been pursued considering three distinct aspects.

(i) Software Development Tools: They consist of the programming languages, text editors, macroprocessors, interpreter/compiler, file system, linkage editor/linker, program verifier, data flow analyzer, test data generator, symbolic executors, dynamic frequency analyzer, source program optimizer, and source program controller. It should be noted that these tools are needed to assist in the creative parts and possibly automate the noncreative parts of the software development.

(ii) Management Techniques: This is to manage the activities of the personnel and to monitor their progress.

(iii) Design Methodologies: These are to attack the particular problems found in large softwares. These methodologies, to a great extent, have led to the development of supporting language facilities.

The clear identification of the above three aspects is important because a large software development is not merely a matter of programming, but a management problem. Thus, there must be capabilities of estimating the necessary resources, assigning resources at appropriate times, subdividing the development effort into phases, monitoring these phases and imposing a set of standards for each activity. The details of all the above aspects are out of the scope of this paper. However, we will consider the programming language as a tool for development of large softwares and will discuss the related issues.

The basic goals of a software development process are: smooth design, reliability, efficiency and maintainability of the software. The programming languages must provide tools for assisting in achievement of these goals. In the subsequent chapters, we discuss the large software development goals, objective criteria for the language choice and various language features which satisfy these criteria. In the last two chapters we

consider two modern languages, Pascal and Ada as software-development tools. Selected features of these languages are presented to demonstrate how each one of these does or does not meet the objective criteria for the language choice.

CHAPTER 2

SOFTWARE DESIGN AND PROGRAMMING LANGUAGES

The programming language constitutes one of many tools required for software development. Thus the programming language's merit must be judged on how it affects the software development and how well it can be integrated with other software production tools.

The software production process consists of several sequential phases. Each phase involves a distinct activity and results in a clearly identifiable product. A programming language should support all the phases of the development rather than only coding.

1. Requirements Analysis Phase:

The real needs of users should be specified. Also a look at perceived needs should be taken. Then requirements to be satisfied by the software system should be specified. The success of software depends upon each of these stages reflecting the previous one. The results of this phase should be documented in the form of a "Requirements Document". This should include the cost and performance analyses, the user's manual and what the system is to do. The people charged with the responsibility should interact with the programming language facilities for keeping track of requirements and

guarding against inconsistencies. They should keep track of documentation as changes are being made.

2. Software Design Specifications:

Based on the requirements document, a "Specifications Document" should be produced, which should include:

(a) the identification of modules constituting the software system and their interfaces,

(b) the identification of design methodology, and

(c) the test data.

The programming language can aid in all these. Interaction between the programming language and design specifications is two-way. Based on the above factors, a suitable choice of language can be made for implementation. Many times an available programming language feature influences the decisions about the above items. For example, Ada package facilities may dictate the choice of "information hiding methodology". (details later).

3. Implementation:

A programming language is directly connected with implementation of the software and documentation. The choice of programming language should be such that its facilities are compatible with the methodology identified in the previous phase.

4. Certification:

The purpose of this phase is to ensure that the software meets the stated requirements. It involves the following steps.

(a) Testing

(i) Module Testing: testing individual modules to ensure that interface specifications are met.

(ii) Integration Testing: for detecting intermodule inconsistencies.

(iii) System Testing: final testing to ensure whether the software system is according to the requirements.

(b) Program Verifications: This concerns assessing the correctness of the program and is usually done by formal axiomatic proofs.

The programming language should help by providing appropriate facilities for collecting test data from other phases and ensuring currency of tests. It should support formal specification and verification of modules. The programming language should also have facilities for tracing changes to specifications, test data and documentation, identifying all affected areas easily.

Software Development Process and Goals:

A software system, when fully developed and certified, is complex and costly. Hence, certain design goals should

be kept in sight. We mention below three important goals which form the basis for the rest of our discussion in this paper.

1. Reliability:

   (a) The user should feel comfortable to use the software inspite of infrequent hardware, software and system failures or anamolies.

   (b) Software should be correct so that it behaves according to specifications.

   (c) Rigorous and unambiguous specifications should be provided to prove correctness convincingly.

2. Maintainability:

A software system is a huge and complicated program entailing enormous cost in terms of human efforts and money. Therefore, it cannot be thrown away if requirements change or latent errors are detected. Hence, it is much more economical to modify the software system. Consequently, tools should be provided to modify and maintain it.

3. Efficiency:

Software should be efficient in terms of execution time and storage space. It should also be efficient in terms of development and maintainability efforts.

CHAPTER 3

OBJECTIVE CRITERIA FOR PROGRAMMING LANGUAGE CHOICE

There are certain language qualities which support the development, reliability, efficiency and maintainability of the software. At the development level the programming language should aid the programmer in design, documentation and debugging. Usually the whole program is divided into simpler subtasks, each with clear purpose and interface. The programming language should aid in expressing each level from overall strategy to the code. It should also enforce conventions for harmonious interfaces between tasks.

The documentation is for readability so that adaptation to changes and removing latent errors can easily be done. It should be an integral part of the design. The programming language should support self-documenting codes displaying a pleasant style.

Features supporting debugging are very important for producing reliable programs. The detection of all errors should be guaranteed at the compile time. Any left out error should be cheaply detectable at run time. The programming language should be such that its implementation (compiler) is fast and compact, and its object code is efficient so that extra instructions may

be inserted for debugging purposes and several runs at the testing phase may be made without too much cost.

It is clear that for producing an efficient, reliable, and maintainable software, the programming language should have certain qualities. In the following section, we discuss what qualities are necessary for a programming language to be an effective tool for software design process.

1. Writability:

A programming language supports writability if its features facilitate program design by providing constructs that make the specific adopted design methodology more easily expressable. This shifts the programmer's concern from the details of his tools to solving the problem with those tools. Four properties contribute to writability. They are: simplicity, expressiveness, orthogonality, and definiteness.

2. Simplicity:

Natural languages are spoken and utilized easily and without effort once they are learned. In the same manner, a programming language should show easy fluency. Its different features should be easy to learn and remember. The effect of any combinations thereof should be easily understandable and predictable.

The simplicity can be impaired in the following ways:

(i) A programming language may provide several alternative ways of specifying the same concept. For example in C, the addition of 1 to an integer variable can be accomplished in any one of the following four ways:

```
++b; b+=1; b=b+1; b++.
```

(ii) A programming language may allow different concepts to be expressed by the same notation. For example, Ada permits overloading. It should be noted that careful use of overloading can make the program simpler as in the case of Ada where + can also be used for defining the matrix addition procedure. However, ALGOL 68 and Ada both permit the generalization of overloading to user-defined operators and subprograms. This can easily make programs difficult to read.

(iii) Simple features may interact to produce such a program whose behavior is hard to predict. For example, in Ada a behavior that is difficult to predict is that of combining multitasking and exception handling.

3. Expressiveness:

Expressiveness is a measure of how naturally the problem-solving strategies can be mapped into program structures. The various structures and constructs of

language must be able to express the programmer's understanding of the solution path. The rich set of control and type structures found in Pascal give it a high expressive power in spite of it being a simple language without adequate modularization mechanisms. In general, the basic structuring tool such as abstraction of data and control, and modularization directly aid expressiveness.

4. Orthogonality:

Any combination of the basic primitives of the language should be allowed. This results in the greatest possible generality, a situation free from restrictions or special cases. The lack of orthogonality can force the programmer into difficult and clumsy formats. It can cause the program to make excessive manipulations of data or instructions. An example of the consequences of lack of orthogonality can be found in Pascal. Several restrictions exist which include: (1) a file cannot be passed by value, (2) components of a packed data structure cannot be passed by reference, (3) procedures and functions passed as parameters can only be by values, (4) the type of a formal parameter can only be specified by a type identifier and not by its representation. As an example of this, consider the following:

(i) procedure ILLEGAL(var X: array [1..4] of real; Y:char);
(ii)procedure LEGAL(var X:T;Y:char); where T is declared in an outer scope as type T = array [1..4] of real;

The declaration (ii) is legal whereas (i) is not.

5. Definiteness:

This criterion is concerned with the accuracy of a programming language's description of its syntax and semantics. This is intended to reduce vagueness which can be accomplished by standard definitions and minimizing ambiguously stated features. A trade-off exists between readability and definiteness such that a fairly readable language like Pascal may allow incorrect programs, while a grammatically precise language such as ALGOL 68 may be very difficult to read and utilize. For example, the grammar of Pascal does not specify that an arithmetic expression, such as x + 4.75, cannot be assigned to a boolean variable. Thus, syntactically correct programs are legal in Pascal only if they also satisfy an additional set of conditions stated in English words in the report. The report also does not deal with procedures in generality and leaves out type compatibility (details in the next chapter).

6. Readability:

Readability is a major factor which influences program modifiability and maintenance. Utilization of

self-documenting code, abstractions of data and control structures, and program modularization contribute to readability. Integrated program documentation in terms of internal comments embedded within the program contribute to readability. Efficient comment delimiters, which reduce the likelihood of omitting an ending delimiter improves the use of comments to the programmer. In stream oriented languages, such as Pascal, comments are placed within special delimiting characters and can appear anywhere within the program. An unfortunate consequence of this is that when the programmer forgets a trailing delimiter the translator ignores entire pieces of the program up to the next delimeter. The choice of a line oriented language eliminates this problem because the end of the line is taken to be an implicit delimiter. We find that the Pascal comment convention is risky whereas Ada's convention is safe because the comment is delimited by the end of the line Ada is also easier to read. Lexical conventions such as identifier length, punctuation, use of spaces, etc. also affect readability by giving a more natural and segregated appearance to terms. FORTRAN and C place severe restrictions on the length of identifiers which can force the programmer to virtually encrypt his/her identifiers. The capability to use underscore characters in identifiers, which is permitted in Ada, can

enhance readability. For example, "clientname" as contrasted with "client_name" is less readable.

The syntax of the language can influence the program readability. Explicit delimiters such as if..fi, do..od, etc. should be preferred to the begin..end pair because they indicate the purpose of statements. Suitable indentation convention can make the program more readable.

The semantics of the language can also affect the program readability. For example, parameter passing by reference and global variables can produce aliasing in Ada. Languages like Euclid impose semantic restrictions that make aliasing illegal and hence detectable.

The readability is very crucial in production of large softwares when a group of programmers are involved. It is important to reduce the amount of individual variations so that one member can read the program written by another member.

7. Reliability:

A reliable program manipulates information in the way the programmer intended. Where large software can be developed and validated one module at a time, as in Ada, the separate compilation facilities enhance the reliability of the program as a whole. Formal definitions of program semantics aid the program reliability by favoring program verification. Additionally, the

reliability depends on the reliability of the language implmentation by a compiler.

Rigorous distinction between static and dynamic checks can enhance reliability in so much as the programmer is aware of the degree of validation at each step of processing. Ada makes this distinction whereas Pascal does not.

Another factor influencing reliability is modifiability specially during maintenance. We discuss modifiability separately below.

8. Modularity:

Modularity is generally considered to be the only effective strategy for developing large softwares. Modules have been defined on the basis of the size of the program units which constitute a software system. The concept of independence is an alternative means of defining program modules. The goal of independence is to have each program module to be understood and implemented independently of the other modules in the system. Each module would implement a single and simple conceptual function. Modularity has been called "purposeful structuring." Ada, CLU and Modula-2 support modularity via packages, clusters and modules respectively.

9. Security:

A language supports security if all type checks can be performed statically (at compile time). A few excepted ones must be checkable at run time without too much cost. This is important for assigning storage efficiently and the speed of the object code. This directly aids the reliability and efficiency of the software. Strongly-typed languages such as ALGOL 68 and Ada support this feature, whereas Pascal does not. Explicit discussions of this feature, in cases of Pascal and Ada, are provided in last two chapters.

10. Modifiability:

Modifiability aids reliability of the software, because during the maintenance phase we must be able to modify the program while retaining its reliability. The language syntax is basically responsible for the ease of the program modification. As seen before ALGOL 68 and Ada delimeters are preferable to begin...end of Pascal. For adding a new statement to a single statement else-branch requires a bracketing pair of begin and end. In case of nondeliberate omission, incorrect program is produced even though the program is syntactically correct.

Other factors influencing modifiability are typed names and various abstractions. These enable the changes to be localized.

CHAPTER 4

DESIRABLE LANGUAGE FEATURES FOR SOFTWARE

Several programming languages are available for software development and production. In this chapter we discuss explicit language features which satisfy the criteria mentioned in the previous chapters in regard to producing a reliable and maintainable software.

1. Good Syntax:

The good syntactic forms are required, for a lot of time is spent in lexical and syntax analyses. It helps the compiler pinpoint error, diagnose causes, recover and move on. As seen before, it is important for supporting readability, reliability and modifiability.

2. Strongly-typed Variables:

The language is said to be strongly typed if all type checkings can be done statically at compile time. There are obvious advantages to this.

(1) It provides security by static checking.

(2) It aids maintainability by supporting modifiability.

(3) It aids writability.

(4) It aids readability and prohibits undisciplined manipulation of underlying representation.

(5) It provides for accuracy checking by helping compiler to enforce precision (e.g. Fixed point number in Ada).

(6) It provides for efficient implementation.

One example of strongly typed language is ALGOL 68.

3. Block Structure:

Blocks are useful because scope and locality are associated with them. Users can use them to manifest program structure. There is no danger of interference without permission. The space occupied by already executed blocks can be freed. The block structure permits efficient paging in memory management as together-used variables can be declared together, and hence, can be assigned contiguous locations. Space can be saved by overlaying noninterferring blocks.

4. Procedures:

They give all advantages of an extensible language, providing for the storage economy, compact code, and structured programming. The procedures and their advantages are so well-known that the details may be omitted here in the interest of space.

5. Appropriate Levels Of Abstractions:

Abstractions should be at the same level and compatible with each other. "Goto", typeless variables, and pointer/reference are not at the same level as other

abstractions. These should be absent or restricted. Instead of jumping out of loops, while should be used to terminate loops, and computed "Go to" may be replaced by case. Pointers should be severely restricted. Every pointer variable should point to objects of a single type only. There should not be automatic coercion, that is pointers and object pointed to should be distinguished. All these restrictions are satisfied by Pascal pointers.

6. Appropriate Control Structures:

This is essential for structured programming. There are several control structures available in modern languages. "Go to" use should be restricted. For jumping out of a loop an EXIT_type statement (Ada) should be used. The loop-and-half of Dahl (see bibliography) is very appropriate:

```
loop
     statement sequence1
while condition
     statement sequence2
repeat
```

This is a generalization of while (if statement $sequence_1$ is left out) and repeat (if statement $sequence_2$ is left out). Testing is done in the middle. Several languages permit user defined controls. Alphard (forms) and CLU (clusters) are some of the examples.

7. Creation Of New Types:

It helps in the following ways.

(1) Readability: Appropriate choice of new names can help readability.

(2) Modifiability: If the data structure representing a variable is changed, then only this type declaration needs to be changed.

(3) Factorization: A complicated data structure can be coded only once and used several times.

(4) Type compatibility: Type equivalence can be used to treat the compatible types the same as explained below. It should be noted that it directly aids security.

8. Rigorous Type Compatibility:

Type compatibilities are of several kind, out of which the following two are the most important.

(1) Name-Equivalence: Two variables are compatible when they have the same type names or appear in the same declaration.

(2) Structural-Equivalence: Two variables are compatible if they have the same structures.

Example:

```
type t = array [1..20] of integer;
Var a, b : array [1..20] of integer;
       c : array [1..20] of integer;
```

```
d : t;
e : t;
```

In this example a and b, and d and e are compatible under name equivalence. However, a and c are not as they do not have same type names. Since a, b, and c have the same structures, they are compatible under the structural equivalence.

Name equivalence has advantages over the structural equivalence because of the following reasons. It is easy to implement as the latter requires pattern matching. It is closer to the abstract data type. Moreover, the named types aid readability in so far as they indicate commonality of properties specified by using the same type names.

9. Data Abstraction:

The language should have features so that the user can define abstract data types as in Ada (packages) or CLU (clusters). This directly aids readability, modifiability, factorization, checking type-compatibility and extensibility.

10. Exception Handling:

Exception is a condition which is detected and brought to the notice of the invoker when an attempt is made to execute an operation. The response of the invoker is called handling of the exception.

The exception handling facility renders the operation more general and provides interaction between two levels of abstractions (invoker's and of unit where exception resides). The exception handling may be implemented for indicating error conditions, classifying valid results (e.g., signaling that the remaining storage is critically minimum), monitoring the intermediate results, and demanding information which may be costly to compute a priori. The details of exception handling are out of the scope of this paper. However, it should be noted that it is an important feature of the language supporting the reliability of the software as it handles anamolies and errors gracefully as mentioned earlier.

There are several languages with exception handling capabilities. PL/I, CLU and Ada are notable. We will discuss the Ada's exception handling in details later.

11. Concurrent Processing Facilities:

A software such as system program may have concurrent routines. For this, facilities of concurrent processes (tasks) are required. The issues involved here are task's definition, start and termination two tasks' synchronization, and communication. At least two languages provide this facility: CSP and Ada. The further development of the subject is beyond the scope of the paper. However, details of Ada's tasks will be provided later.

CHAPTER 5

PASCAL AS A SOFTWARE DEVELOPMENT TOOL

According to Wirth, its designer, Pascal[1] is a tool for producing a reliable and efficient software. This is because of its conceptual simplicity, efficient implementation, and features for making programs transparent and avoiding and detecting mistakes. It provides the compiler with redundant information for checking consistency without too much overhead. How Pascal does all these is explained below.

Pascal has a rich assortment of types: symbolic scalar, record set, subrange, etc. It also contains simple iteration and selection statements, such as while, repeat, for, if-then-else and case. Each variable has a certain type so that declarations are statements of invariant properties of variables. This feature is essential for efficient automatic program verification and implementation, and for reliability of software. Pascal has a rich set of built-in data structures in forms of arrays, records, lists, and sets. It also has program units such as procedures. All these support modifiability

---

[1]The version used in this paper is Wirth's original version described in "The Programming Language Pascal," Acta Informatica 1 (1971).

and maintainability of the software.

Although Pascal is one of the very successful languages, it has serious problems as a tool for the development of a software with a high degree of reliability and maintainability. In the following we point out some of its shortcomings in relation to the criteria set forth in previous chapters.

(1) No Block Structure: Pascal has no block structure, as ALGOL 60 or Ada.

(2) No Own Variable: Own variable is declared locally and is shared by other units where implementation details are hidden. An example is Ada package and its variables. The relevance of this feature is underscored by noting that it aids readability and modifiability of the software. The own concept is basic to the structure and understanding of co-routines and concurrent processes.

(3) No Initialization At Declaration: References to uninitialized variables are difficult to detect and all hardware mechanisms known for this purpose are quite expensive. A possible solution is to require that every variable be initialized when it is declared or that every variable be assigned in such a way that the compiler can easily verify that there are no references to uninitialized variables. Both

these possibilities are absent from Pascal.

(4) Pascal Is Not Strongly Typed: The following violations of strong typing are directly against security.

(i) Procedure Parameters Not Secure: The Pascal language allows procedure declaration such as:

```
procedure f (procedure g);
```

The parameter types of g are not known at compile time.

(ii) No Static Subrange Checking: Suppose a and b are in subrange 1..10, then it cannot be checked statically that a + b is in 1..10. It has to be done at run time.

(iii) No Rigorous Type Compatibility Concept: In fact, in the original report there is no mention of the concept at all.

(iv) Use Of Variant Records: The correct use of variant records cannot be checked statically and run-time checking is seldom provided because of severe impact on execution effeciency. So the programmer has a loop hole to short-cut type protection. Pascal allows variant records with or without tag fields.

```
TYPE DEPARTMENT=(HOUSEWARE,SPORTS,DRUGS,FOOD,BEVERAGE);
  MONTH = 1..12;
  ITEM = RECORD PRICE: REAL;
```

```
CASE AVAILABLE: BOOLEAN OF
  TRUE: (AMOUNT: INTEGER; WHERE: DEPARTMENT);
  FALSE: (MONTHEXPECTED:MONTH);
END;
```

The identifier AVAILABLE is called the tag field of the record structure. The conventional implementation of variant records consists of overlapping all variants over the same storage area. Consequently, variant records allow the programmer to interpret the bits stored in this area under the different views by the type of each variant. Suppose AVAILABLE is true and variant field has been set so that the field AMOUNT has value v_0. The programmer can set AVAILABLE to FALSE and interpret the v_0 as the value of MONTHEXPECTED. This gives an uncontrolled access to the storage. Still worse happens when tag field (AVAILABLE) is absent. Either AMOUNT and WHERE or MONTHEXPECTED can be used when they are not present. This error is detectable neither at compile nor at run time, because there is no field of the record to denote the currently applicable variant.

(5) Problems With Pascal Pointers:

(i) Dangling pointers: When an object pointed to is removed, the pointer is left dangling. The Pascal

language does not have facilities to handle this insecurity.

(ii) Uninitialized pointers: It can cause uncontrolled access to storage because bit string found in the location bound to the pointer could be interpreted as a pointer value. Addressing with this value cannot be detected even at the run-time.

(iii) Because of undefined type compatibility, serious problems arise with regard to the Pascal pointers. For example, consider the following.

```
TYPE PTR = ↑ NODE; REF = ↑ NODE;
     NODE = RECORD ITEM : INTEGER;
                   NEXT : PTR
     END;
```

It is not clear whether it is legal to assign a variable of type REF to a variable of type PTR.

(6) Inflexibility Of Array Bounds: The array bounds must be declared as fixed. If one had to create a general program to add two vectors of the same but arbitrary sizes, one cannot effectively do it. Modula - 2 and Ada provide the facility of flexible bound declaration. The details of Ada features are discussed in the next chapter.

(7) No Exponentiation Operators: The original Pascal has no exponentiation operator. Conseqeuntly, the

user has to provide his own routine for such an operation.

(8) Buffering Not Hidden: Buffers and buffering mechanisms are not hidden from the programmer in put (f) and get (f) file operations. Reset (f) rewinds the file and reads the first component. This can cause difficulty with physical device because of slow speed.

(9) No Control Abstraction: Users cannot define new control structures as in CLU or Alphard. Consequently, Pascal does not support extensionability.

(10) No Data Abstraction: No effective facility to define new data structure as in CLU or Ada.

(11) No Exception Handling: Pascal does not have system-provided exception handling facilities.

(12) Not Very Readable: Comment conventions are such that a section of program can be ignored when unintentional mistakes are made by omitting *). The statements such as if..fi, do..od, etc. are not present. The lexical convention permits limited possibilities for variable names.

(13) No Concurrent Processes: Because of the lack of this feature, Pascal is not an effective tool for system programming.

(14) Ambiguities In Pascal:

(a) Type-Declaration Order: Although the report does not say so explicitly, the language features appear to favor one-pass compiler and implementors have assumed this to be the designer's intent. However, this implicit one-pass compilation capability creates some traps into which implementors have fallen. Generally, one pass compilation requires that the declaration of an identifier must precede references to it. However, Pascal does not specify it, especially that declaration should occur before use within the type definitions or procedure declaration parts. Pascal imposes rigid order on constant, type, variables, and procedure declarations. Thus, it does not allow natural groupings of the types, variables and procedures which manipulate them. This inability in structuring large programs is one of Pascal's most frustrating limitations.

(b)Set Constructors: According to section 8 of the Report, "Expressions which are members of a set must all be of the same type, which is the base type of the set". Thus the constructor [4, 6, 10..20, 25] has type SET OF INTEGER. But elsewhere in the Report it is mentioned that, "The implementor may

set a limit to the size of a base type over which a set can be defined". We note that since the size of INTEGER is unbounded, one might conclude that the above example is illegal in at least some implementations. In practice there is another difficulty. Even if we suppose that a limit on the size of base type exists, the compiler may assume that intended base type of [4, 6, 10..20, 25] is some subrange of integers, and apply an implicit range to subrange transfer to its members. The difficulty is that the intended subrange is not apparent. This may cause problems for representation of the set.

(15) Range Violations: Accessing an element of a Pascal array for out-of-bound index is insecure. Other contexts in which a value outside a permitted range can arise are: assignments to subrange variables, case selection, set membership creation and testing, and overflow in integer and real arithmetic. Although, all such violations cannot be dealt with by the language designer, the attempt must be made to reduce the cost of run-time detection. Pascal has taken steps towards this goal, but a lot is left to be desired.

It is obvious that although Pascal has many good features, it also has several shortcomings. It does not quite support the design, reliability and maintainability of a large software system. In the next chapter we will discuss Ada and see how it can prove to be a good tool for accomplishing our goals.

CHAPTER 6

ADA AS A SOFTWARE DEVELOPMENT TOOL

Ada seems to be a language which has incorporated all desirable features of modern programming languages. It is built upon Pascal's data and control structures. However, many of the shortcomings of Pascal discussed in the previous chapter have been eliminated. It also contains many desirable features not available in the existing modern languages. Until recently such concepts (e.g. EXIT, loops, concurrent processes) have only been used by researchers as desired features. Our purpose here is not to provide details of the Ada language. We plan to pick those features which satisfy the objective criteria of a language choice for large software production. At this point, we would like to point out that Ada seems to have all required features with a possible exception of simplicity. However, it should be made clear that Ada may violate simplicity only in one sense that it is difficult for a beginner to master.

1. Ada's Type Structures Are Secure:

The following discussion of the type structures of Ada is not intended to be detailed and comprehensive. It is intended to show how Ada's structures overcome the problems and insecurities of Pascal's structures.

(a) We first note that Ada distinguishes between the static and dynamic properties of types. The static properties concern the applicable operations, whereas the dynamic properties are range constraints on integers or index constraints on arrays. To make the distinction clear, the programmer is allowed to specify a dynamic property of a type by defining its subtype. For example, INTEGER is a system-provided type whose subtype can be defined as:

Subtype NATURAL is INTEGER range 1..INTEGER'LAST;

A variable of subtype NATURAL inherits all the properties from the type INTEGER except that it satisfies certain constraint: it belongs to the subset {1,2,3,....} of integers. The constraint may involve expressions that cannot be evaluated at compile-time (statically). This is done when subtype declaration is elaborated at the entry of the scope where declaration appears. However, the programmer is aware of it and has full information about the degree of validation. This is crucial for reliability as seen earlier.

(b) Ada distinguishes between static and dynamic arrays. We first note that Pascal does not have dynamic arrays at all so that changing the bounds of arrays implies recompilation of the program. In Ada's

dynamic array, the type is characterized by the type of components, the number of indices, and the type of each index. The bounds are not considered as a part of the array type. For example, consider the following.

type VECTOR is array (INTEGER <>) of FLOAT;

where, <> stands for unspecified range and FLOAT is system-provided type. The objects of type VECTOR have components of type FLOAT indexed by values in unspecified range of integers. But for any object of an array type, the bounds of each index must be specified:

```
subtype VECTOR_6 is VECTOR (1..6);
subtype VECTOR_5 is VECTOR (1..5);
```

Then the variables can be declared as

```
A, B: VECTOR_6; C, D: VECTOR_5
```

(c) Ada also provides another mechanism (initialization) for installing bounds. Suppose B=(1, 2, 1.5, 0, 1) is of type VECTOR_5. Then we can declare C:constant VECTOR:=B. However, it should be noted that C denotes read-only data because of constant specification.

(d) Ada also provides the bounds instantiation via parameter passing. To illustrate the point, consider the following example.

Suppose we intend to add two same-size vectors. This can be accomplished by writing the following Ada function.

```
function ADD_VECTOR (A, B: in VECTOR) return VECTOR is
   W:VECTOR (1..A'LENGTH);
      begin
         for I in A'range loop W(I):=A(I)+B(I);
         end loop return W;
      end ADD_VECTOR;
```

We note that W is local to the function. The loop variable I does not need explicit declaration. The formal parameters A, B automatically have the same bounds as the actual parameters. Such bounds are automatically accessible via A'range. This renders the function completely general which can be called with arrays of different sizes as parameters.

(e) In the following we show how Ada's variant records are secure as opposed to the Pascal's. Let us consider the variant record described in the previous chapter to demonstrate its security in Ada.

```
type DEPARTMENT is (HOUSEWARE,SPORTS,DRUGS,FOOD,BEVERAGE);
subtype MONTH is INTEGER range 1..12;
type ITEM (AVAILABLE: BOOLEAN:=TRUE) is
     record PRICE: REAL;
            case AVAILABLE of
```

```
            when TRUE => AMOUNT:INTEGER,
                         WHERE:DEPARTMENT;
            when FALSE => MONTH_EXPECTED: MONTH;
        end case;
    end record;
```

It is possible to define subtype NOT_AVAILABLE is ITEM (FALSE) where variant is frozen. Ada record has tag field AVAILABLE defining the possible variant of ITEM. The tag is mandatory and cannot be assigned directly. In case the tag AVAILABLE is not initialized (as above) the tag constraint must be provided for any object declaration such as A : ITEM (FALSE). The tag value can be changed only for objects not explicitly constrained, and by assignment to the record as a whole. Consider the following example to illustrate the point. Suppose I : ITEM; where ITEM has initial value TRUE. Then it is legal to write I:=A. Here the variant is set to FALSE because A is of type ITEM(FALSE). The example of the whole record assignment is:

```
I:=(PRICE=> 2.50,AVAILABLE=> TRUE,AMOUNT=>1000,
                    WHERE=>BEVERAGE);
```

If an access in attempted to a component (say) I·WHERE then the compiler automatially converts it into the run-time test preceding the access:

```
if not I·AVAILABLE then raise CONSTRAINT ERROR end if;
```

2. Ada Has Secure Pointers:

The example of an Ada pointer is as follows.

```
type BUFFER is
  record
    MESSAGE:STRING(1..8);     -- STRING System defined
    PRIORITY: INTEGER range 1..100;
  end record,
type POINTER is access BUFFER;
P1, P2, P3:  POINTER;
```

We can create new objects:

```
P1:= new BUFFER;     -- create a BUFFER-type record
P2:= new BUFFER'(MESSAGE => "DELETE",PRIORITY => 10);
P3:= new BUFFER'(MESSAGE => "READY",PRIORITY => 1);
```

Two pointers can refer to the same object if their types are compatible under name-equivalence (see later). This was seen to be ambiguous in case of Pascal. Two pointers of noncompatible types are guaranteed not to point to the same object.

3. Ada's Control Structures:

Ada incorporates all desirable control structures with pleasant syntactic conventions directly supporting readability, expressiveness and reliability.

(a). Selection:

(i) if C_1 then S_1 end if;

(ii) if C_1 then S_1;else if C_2 then S_2 end if;end if;

(iii) if C_1 then S_1;elsif C_2 then S_2 end if;

(iv) Case x is

when a_1,...a_2 => S_1;

when b => S_2;

when others => S_3; -- out of bound

-- cannot happen here

end case;

(b). Counter-Driven Iterations:

(i) for i in 1..50 loop S end loop;

(ii) for i in A'range loop S end loop;

(c) Condition-Driven Iterations:

(i) while C loop S end loop;

(ii) loop S_1 exit---S_2 end loop; -- infinite loop

(d) Jump:

Labels <<L>>; goto L

"Goto" are probably not essential in Ada and we could use other structured constructs to implement algorithms. If we must use "goto", we limit its scope. Ada restricts the scope of "goto" in the following manner. Execution of a "goto" may not transfer control

(i) into a compound statement namely if, loop, accept, case, block or exception handler.

(ii) from parts of one if, case, select, or exception handler to another.

(iii) back into a block, subprogram, task or package body.

In general it may transfer control only within the same lexical level.

We see that Ada supports highly structured programming, and hence, expressiveness. The selection and iteration constructs contain delimeters so that the Ada programs are highly readable.

4. Ada's Subprograms:

(a) Functions: The example has been given in 1(d) above

(b) Procedures:

```
procedure INNER_PRODUCT(A,B:in VECTOR,IP:out FLOAT)is
  SUM:FLOAT:=0.0;
  begin                    -- type VECTOR defined earlier
    for i in A'range loop SUM:=SUM+A(i)*B(I)
  end loop;
  IP:=SUM;
end INNER_PRODUCT;
```

We note that Ada procedures are very general. The above procedure can take any dimension vectors A, B and find their inner product. Such generality is not possible in languages like Pascal. The local variables and parameter can be initialized at declaration. This completely removes the insecurity of referencing uninitialized variables. The parameters can be overloaded, providing orthogonality and generality. Ada procedures have many more modern features.

For the details Ada report may be consulted. We conclude by remarking that a procedure can be passed as a parameter only when its parameters have been declared. This provides for static checking of correct use and eliminates insecurity seen in Pascal.

5. Ada's Package Facilities:

(a) Packages can be used for declaration of a set of common entities such as variables, constants, and types. For example, consider the following package declaration.

```
package COMPLEX_NUMBER is
   type COMPLEX is
     record RE:INTEGER;IM:INTEGER;
     end record;
     LIST:array (1..1000) of COMPLEX;
end COMPLEX_NUMBER;
```

The processing of the package is such that it acts as a declaration of the described variables and types. The variables and types can be used in two ways.

(i)
```
with COMPLEX_NUMBER
Type A:LIST;
```

(ii)
```
Type A:COMPLEX_NUMBER.LIST;
```

The general structure of a package consists of the package specification and package body. The specification part contains all "exportable" information. The package body

contains hidden details of implementation and initialization, and is executed upon activation of the unit using the package declaration. The details are provided in (b) below

(b) Packages can be used for grouping related subprograms. For example, consider the set of vector operations.

```
package VECTOR_OP is
 subtype INDEX is INTEGER range 0..INTEGER'LAST;
 type VECTOR is private;
 function"+"(u,v:in VECTOR)return VECTOR;
 function"*"(x:in FLOAT,u:in VECTOR)return VECTOR;
private
 type VECTOR is array(INDEX range <>) of FLOAT;
end VECTOR_OP;
package body VECT_OP is
  function"+"(u,v:in VECTOR)return VECTOR is
     W:VECTOR(1..u'LENGTH);
     begin
         for i in u'range loop W(i):=U(i)+V(i);
         end loop; return W;
     end "+";
  function"*"(x:in FLOAT,u:in VECTOR)
                    return VECTOR is
     W:VECTOR(1..u'LENGTH);
```

```
        begin
            for i in u'range loop w(i):=x*u(i);
            end loop;return W;
        end"*";
    end VECT_OP;
```

The type VECTOR and operations + and * are exportable. The package body is hidden from outside. Use of the +, * for vectors is called overloading these arithmetic operations. This provides mathematical appearance to these vector operations. It should be noted that type VECTOR exported by the module is private. This means that the details of type representation enclosed between private -- end VECTOR_OP in the specification part is not visible outside the package. The effect of this restriction is that variables of type VECTOR can be manipulated only by operations + and * provided by the package. The assignment and equality/inequality test, which are predefined are also permitted. These latter operations can be curbed using limited private type. One good point about Ada packages is that they are automatically created when units declaring the variables of type VECTOR are activated. This is a definite advantage over CLU'S clusters where the explicit create operation is required. The private type definition is included in the specification part because the

specification and body are separately compiled and compilation of using unit depends on specification and not body.

(c) Packages can be used for creating new abstract data structures. The example, 5(b) provides a new data structure namely VECTOR. Package facilities can be used to create higher level structures such as stacks, queues, etc.

6. Generic Facilities: Consider the following example.

```
generic
  type ELEMENT is private;
     package SET_OP is
       type SET is private;
     procedure INSERT(x:in out SET;y:in ELEMENT);
     procedure DELETE(x:in out SET,y:in ELEMENT);
     function MEMBER(x:in SET;y:in ELEMENT)
        return BOOLEAN;
  private
     type SET is array(1..50) of ELEMENT;
  end SET_OP;
```

The details of the package body is left out. Since ELEMENT is of private type, only operations (by default) permitted on ELEMENT-type variables are: assignment and testing equality/inequality. Before using the generic package, it must be instantiated:

```
    package INTEGER_SET_OP is new SET_OP (INTEGER);
```

We observe that this facility directly supports, factorization and generality.

7. Ada's Exception Handling:

Ada supports reliability of the software via exception handling. Ada's exception handling can be demonstrated by the following example:

```
package DEMONSTRATE is
   ERROR: exception;
   procedure Q (x, y: in INTEGER, z: out FLOAT);
end DEMONSTRATE;
package body DEMONSTRATE is
   procedure Q (x, y: in INTEGER; z:out FLOAT) is
      begin
         if y=0 then raise ERROR;     -- procedure Q is the
                                      -- raising unit
           else z:=x/y;
         endif;
end DEMONSTRATE;
```

Now user writes the following block.

```
    BLOCK          -- BLOCK is block name.
                   -- Block body is enclosed between
    declare        -- declare and end BLOCK. Details later
```

```
    begin
      use DEMONSTRATE;
      Q (a,b,z);
      exception                -- exception handler
                               -- resides here in the BLOCK
      when ERROR =>            -- do something
      when Others =>           -- do something
    end;
  end BLOCK;
```

Any program unit such as a procedure, block or package can raise exception (say) ERROR by instruction raise: ERROR. Handler can reside within the raising or enclosing blocks or outside. If it resides within the same block, then it skips until the handler body executes it, and then the raising unit is terminated. If it is outside the raising block, then the raising unit is terminated and raising is propagated to the enclosing block. We note that there are no parameters. So the raising unit can contain the handler where user specified clean-up can be performed before returning to the caller.

8. Ada's Communicating Processes:

Such processes in Ada terminology are called tasks. This is an effective mechanism to implement concurrent routines. The necessity of such facilities has been explained earlier. We begin with an example.

```
task A is
entry P(x: in FLOAT);
end A;
task body A is    -- a does not know the name (B) of caller
begin             -- y below is a global var
accept P(x:in FLOAT) do y:=x*x end;
                  -- this accept point must rendezvous with
                  -- entry call in task B for communication
end A;
task B is
entry Q (x:in FLOAT);
end B;
task body B is          -- task B owns task A
                        -- so it knows the same of the called
v: FLOAT;
begin
v:=A.P (12.0);          -- entry call for A made here and
                        -- must rendezvous with accept in A
accept Q(X:in FLOAT)do z:=x+4.0 end;    -- z is a global var
end B;
```

The entry call differs from a procedure call in that the entry is executed by the owning task only. Note that entry call is made in task B by v:=A.P(12.0). Upon this call, in task A the statement "accept P (x:in FLOAT) do y:=x*x" is executed. In other words, entry call in B (for

A) and accept in A's body are synchronized (rendezvous). Thus if entry call in B is reached before accept in A's body then B waits for A to reach accept point for rendezvous. If A reaches accept and no entry call yet has been made, then A waits until B's entry-call point is reached. After rendezvous the exchange of messages takes place (i.e., accept of A is executed) and then A and B both proceed in parallel. It should be noted that calling task (B) knows the name of called (A), but A does not know the name of B. This differs from CSP, where both know the name of each other. Other details of tasks are omitted because of the limited scope of the paper.

9. Ada's Separate Compilation Facilities:

Ada recognizes the need to be able to compile a program in distinct pieces, although we may do otherwise if so desired. The text of the program may be presented in one or more compilation units. The compilation units are:

(a) subprogram declaration

(b) subprogram body

(c) package declaration

(d) package body

(e) generic declaration

(f) generic body

(g) subunits (see topdown approach below)

Tasks are not treated as a separate compilation units. Instead, tasks occur inside packages. For using a task from the library the corresponding package must be in the library. All units mentioned above have been explained earlier. Subunit mentioned in (g) is explained below in topdown approach.

The compilation units of a program are said to belong to a program library in as much as they can be compiled and stored in the library. Any program submitted for compilation is treated in context of package STANDARD which is predefined, precompiled and stored in the library. Consequently, all names of STANDARD are visible automatically. All other precompiled units in library have to be explicitly made visible. Direct visibility may be achieved as follows:

Use clause: Use may be used to gain direct visibility of selected compotents of a package. Suppose a package P has been defined which has procedures Q_1 and Q_2, and a unit wishes to use Q_1. This may be achieved in two ways:

(a) Use P: All names in P are now visible from this point on in the using unit, and hence Q_1 is visible in particular.

(b) P. Q_1 : Only Q_1 is visible.

The advantage of use facilities is that only needed units can be selectively made visible. In general, there

are two approaches provided by Ada for separate compilation.

(a) Bottom-up Approach:

A unit being compiled may refer to a unit in the library. We may use a with clause for all library units including generic. The order of compilation and recompilation among the program units is determined by references to library units in context specification. Suppose a package COMPLEX_NUM is a library unit (already compiled and stored). Then the procedure to solve a quadratic equation using this package may be as follows:

```
with COMPLEX_NUM;use COMPLEX_NUM;
   procedure QUADRATIC_ROOT(a,b,c,:in REAL;
                            r1,r2:out COMPLEX)is
   begin.....end;
```

"with COMPLEX_NUM" in the above example dictates that COMPLEX_NUM be compiled first. Thus QUADRATIC_ROOT is dependent on COMPLEX_NUM. Recompilation of a unit forces recompilation of all dependent units. Although the specification and body units of a package are compilable separately, the latter is dependent on the former implicitly (without using with). Any unit using a package depends on its specification unit only and recompilation of the body does not require the

recompilation of the unit. Because of this independence, the private type definition mentioned earlier is in the package specification and not in the body.

(b) Topdown Approach (Subunits):

As an example consider the package body for stack operations.

```
package body STACK is
  S:array(1..100)of REAL;
  Top:INTEGER range 0..100;
  procedure PUSH(x:in REAL) is separate;
  procedure POP(x:out REAL) is separate;
  begin TOP:=0; end STACK;
```

Two subunits PUSH and POP can then be separately compiled and take the form:

```
separate (STACK)
   procedure PUSH(x:in REAL)is
      begin TOP:=TOP+1;S(TOP):=x;
      end PUSH;
separate (STACK)
   procedure POP(x:out REAL)is
      begin x:=S(TOP);TOP:=TOP-1;
      end POP;
```

Note that STACK is dependent on PUSH and POP so that these units must be in library before STACK can be compiled.

10. Ada's Blocks:

A block statement permits textual encapsulation of a sequence of statements, along with local declarations and possibily exception handler. Data resources are allocated only when the block is entered. Ada's data local to the block is not visible outside. Block is limited in scope to its immediate lexical level and cannot be called. This is unlike a procedure which is callable by any other unit. The scope of an object declared in a block extends from the point of its first occurrence to the end of the block. Blocks may be used for encapsulating a section of code for a local exception handler. They can also be used to declare local objects or types that are not to be used by other program units. The Ada's block forms can be illustrated by the following block (SWAP).

```
SWAP
   declare
     T:REAL;
     begin
       T:=A;
       A:=B;
       B:=T
end SWAP;
```

For further details the Ada report may be consulted. We remark that Ada blocks have general visibility and scope rules as discussed earlier.

11. Ada's Strong Typing:

Ada is a strongly typed language. It enforce name-equivalence for the type compatibility, supporting security to the fullest level.

(i) Two subtypes of the same base type are compatible provided they have same constraints.

(ii) Pointers are managable and secure as seen before.

(iii) Variant records are secure because of tag field constraints,

(iv) Procedure parameters must have their parameter declared.

12. Declaration Initialized:

We can declare objects with default initial values, such as:

```
EXTENT : DISTANCE := 0.0;
```

We must initialize objects before they are used because Ada does not define initial values for objects. The result of attempting to use an object before it is given a value will lead to program error. Other initializations instances have been described in foregoing examples. This facility provides for security against accessing uninitialized variables by permitting defaults through initialization.

13. Exponentiation Operators:

Ada is designed to omit the exponential to a real

power. We must write an appropriate overloaded function if real power is desired. It is predefined only for integer powers. But we may not raise an integer to a negative power, since the result would not be of an integer type. Such an attempt will cause a program error.

CHAPTER 7

CONCLUSION

Although Pascal has many nice features, it falls short of being an efffective tool for a large software development. On the other hand, Ada via its features supports the software production goals of design, efficiency, reliability and maintainability. Ada supports readability because of its comment conventions, delimiters in its control statements, good lexical conventions and various type of abstractions. It supports expressivenees through its rich data, control and unit structures. Ada supports orthogonality/generality by permitting overloading of arithmetic operations and parameters, creation of new types and generic facilities for procedures and packages. Ada supports modularity through its package facilities, separate compilation facilities of various units and harmonious interfaces via, separate, use and with. Ada supports reliability by permitting writability and exception handling. It supports efficiency of software by enforcing type security. Ada's softwares are maintainable as it supports factorization, modifiability and modularity. Ada's facilities of tasks help in producing software where concurrent routines are

needed. So we conclude that Ada meets the objective criteria of a language for a large software production where Pascal does not.

BIBLIOGRAPHY

Booch, G. Software Engineering With Ada. Menlo Park, Ca.: Benjamin Cummings Publishing Co., 1983.

Dahl, O. J.; Dijkstra, E. W.; and Hoare, C. A. R. Structured Programming. England: Academic Press, 1972.

DeRemer, F., and Kron, H. Programming-in-the-Large Vs Programming-in-the-Small. IEEE Transactions on Software Engineering SE-2 (1976): 80-86.

Ghezzi, C., and Jazayeri, M. Programming Language Concepts. New York: John Wiley and Sons, 1982.

Horowitz, E. Practical Strategies for Developing Large Software Systems. Menlo Park, Ca.: Addison-Wesley Publishing Co., 1975.

Metzger, P. W. Managing a Programming Project. Englewood Cliffs, N. J.: Prentice Hall, 1973.

Wirth, N. "On the Design of Programming Languages," Information Processing 74, (Proc. IFIP Congress 74). Amsterdam: North-Holland Publishing Co., 1974.

Wirth, N. "The Programming Language Pascal," Acta Informatica 1 (1971): 35-63.