Francis D' Cruze
CSIS 550-01

Pascal Programming Language

Introduction:

Ever since the invention of Charles Babbage's difference engine, computers have required a means of instructing them to perform a specific task. This means is known as a programming language. Computer languages were first composed of a series of steps to wire a particular program; these morphed into a series of steps keyed into the computer and then executed. There is a myth that Ada Byron was the first programmer. Ada Byron was the daughter of English poet Lord Byron, she was born on December 10, 1815 in London (Class Notes). Byron, friend of Babbage, showed her understanding of the concept of a programmed computer in 1842, when she translated from French and annotated a paper by the Italian engineer Luigi F. Menabrea on Babbage's Difference Engine (William's 184). She also added comments and provided examples of how Babbage's engine could be programmed to solve problems (Class notes). Components of Byron's work remain in the modern digital electronic computer that receives a set of instructions, then carries out those instructions.

There are many different types of programming languages. Some are directly understandable by the computer and others that require intermediate translation steps. We can divide these programming languages into three general types: (a) Machine Language, (b) Assembling Language, and (3) High-level language. Machine language is the 'natural language' of a particular computer. Any computer can directly understand only its own machine language. Machine language consists of strings of numbers that instruct

computer to perform their most elementary operations one at a time (Deitel & Deitel, 10). As computer became more popular, it became clear that machine language programming was very slow and tedious for the programmers. Instead of using machine language programmers began to use English-like abbreviation to represent the elementary operation of the computer. This English-like abbreviation is called *assembly languages,* which has translator programs called assemblers that convert assembly-language programs to machine language (Deitel & Deitel, 10).

With the help of *assembly language*, computer usage increased rapidly. Even though assembly language is faster than the machine language to program, still assembly language is very slow. High-level programming languages, in which single statements could be written to accomplish substantial tasks, were developed to speed the programming process. Pascal is a high-level programming language. This (Pascal) language was originally developed by Niklaus Wirth. His principle objectives for Pascal were for the language to be efficient to implement and run, allow for the development of well-structured and well-organized programs, and to serve as a vehicle for the teaching of the important concepts of computer programming. The language was named in honor of the 17th century mathematician and inventor; Blaise Pascal, but this language is not related to him or his work. Pascal was designed in a very orderly approach, it combined many of the best features of the languages in use at the time, COBOL, FORTRAN, and ALGOL. While doing so, many of the irregularities and oddball statements of these languages were cleaned up, which helped it gain users (Wirth, 100-101).

Biography

Before go to the discussion of the language, I like to give the short biography of Niklaus Wirth. In 1958, he (N. Wirth) received the degree of Electronics Engineer from the Swiss Federal Institute of Technology (ETH). In 1960 he received the M. Sc. Degree from Laval University in Quebec, Canada. He pursued his studies toward Ph.D. degree at the University of California at Berkeley in 1963. He was an Assistant Professor at the Computer Science Department at Stanford University until 1967, where he designed the programming language PL360 and ALGOL W (in conjunction with Working Group 2.1). In 1967, he joined as an Assistant Professor at the University of Zurich. Next year (1968) he joined ETH Zurich. There he developed the language Pascal between 1968 and 1970, and Modula-2 between 1979 and 1981 (Bergin, 119-120). Later he designed and developed the Personal Computer Lilith (high-per-formance workstation) in conjunction with the programming language Modula-2, and 32-bit workstation computer Ceres. He also developed the language Oberon, a descendant of Modula-2, which used to design the operating system, Oberon (Bergin, 120) . "He was Chairman of the Division of Computer Science (Informatik) of ETH from 1982 to 1984, and again from 1988 to 1990. Since 1990, he has been head of the Institute of Computer Systems of ETH" (Bergin, 120).

ALGOL

Before I talk about the Pascal Language, I need to discuss shortly about the ALGOL programming Language because ALGOL is the predecessor of Pascal (Bergin, 118). ALGOL stands for ALGOrithmic Language. In between 1957 and 1960, ALGOL was developed by an International Group of Computer People (Clippinger, 17). It was a scientific language rather than a data processing language because it was not concerned

with arrangements of large files of data and the manipulation of those files. ALGOL dealt with the execution of algorithms applies to variables with in the computer (Clippinger, 17). For an example, it was used to solve different mathematical problems such as differential equations.

The ALGOL language took several forms. ALGOL used a metalinguistic formalism (the precise rules of syntax). This meta-language employs four characters which have nothing to do with ALGOL. These are used to enclose names of things about which the metalanguage is talking,::= which means 'is' and | means 'or.' Thus: <digit> ::= 0|1|2|3|4|5|6|7|8|9 is read 'a digit is a 0 or 1 or 2 .. 9'. The 116 basic symbols of ALGOL are the ten digits, 52 letters, upper and lower case, and 52 delimiters (Clippinger, 17).

The six arithmetic operators: + (plus), - (minus), x (times), ÷ (divided by, giving an integer), ↑ (exponentiate), / (divided by).

The six relational operators: < (is less than), ≤ (is less than or equal to), = (equal to), ≥ (is greater than or equal to) > (is greater than), and ≠ (is not equal to).

The five logical operators: ≡ (is equivalent), ⊃ (implies), ∨ (or), ∧ (and), and ¬ (not)

The six sequential operators: Go to, If – then – else, For Do

The seven declarators: Own (user Type), Boolean (which has two values, true or false), Integer (number), Real (number), Array (can be multi-dimensional like a vector or matrix), Switch (designates a choice of 'go to' points), and Procedure (an unit of program that produces certain output from certain input like function) (Clippinger, 17-18)

The program example:

"*Begin integer* n; *real* h, k, pi; *real array* s [0: 180], c [0:180];

Pi := 3.14159; h := pi/180; k := h/2;

S[0] := 0; c[0] := 1;

for n := 0 step 1 *until*

180 *do*

begin comment s[n] is sin n pi/180 and c[n] is cos npi/180; *real* y, z, Y, Z;

integrate: y := s[n]; z := c[n];

Y := y + h * z; Z := z – h * y

S[n + 1] := y + k * (z + Z);

C[n + 1] := z – k * [y + Y]

End integration step

End trig table computation" (Clippinger, 18).

This example show the notion of block, which starts with *begin* and end with *end* keyword. In the beginning of block, there are three declarations (interger, real, and array) which are good for the block only. Variable n is *integer* type numbers, h, k, pi are real numbers, and *array* s, and c are real one dimensional array. Next two lines (from the second line to third line), assign values to the variables and arrays. Next two lines (from forth to fifth) shows the for repetition structure. The *for* statement is self explanatory that 'integrate' will be performed 181 times with n successively equal to 0, 1, 2 … 180. The second block starts with comment (this is a label) which explains what s[n], and c[n] are. These kinds of comments can be used with other statements because they are ignored by the computer. The integration block shows the solution of the differential equation. Last

two lines show that after *end* keyword there are comments, which explain which block is terminating (Clippinger, 18).

Early history of Pascal:

N. Wirth wrote, "The programming language Pascal was designed in the year 1968 – 1969, and I named it after the French philosopher and mathematician, who in 1642 designed one of the first gadgets that might truly be called a digital calculator" (Wirth, 97). In early 1970, the first Pascal compiler was operational and the language definition also was published (Wirth, 97). He wrote, "however, the genuine beginnings date of designing of Pascal was much further back" (Wirth, 97). There were two principal scientific languages: FORTRAN and ALGOL 60 at the beginning of 1960s. Niklaus Wirth joined the Working Group (WG) 2.1 in 1964 (Wirth, 97). The group was responsible for additional development of ALGOL. There were two major groups among the members of the WG. One group was ambitious who did not want to keep the framework of ALGOL 60 and wanted to include features that most of those features were untried and whose consequences for implementors was not proven. Another group was much more conservative that they wanted to keep the body of ALGOL 60 and wanted to add features that were well-understood, and which would increase the area of applications for the successor language (Wirth, 98).

Niklaus Wirth was in the second group who wanted to keep the body of ALGOL 60. He wrote, " I proceeded to implement my own proposal in spite of its rejection, and to incorporate the concept of dynamic data structures and pointer binding

suggested by C. A. R. Hoare" (Wirth, 98). This is called ALGOL W which was implemented at Stanford University for the IBM 360 computer and the out come was published (Wirth, 98). Even though he (N. Wirth) took pragmatic precautions, the implementation was complex, required a run-time support package. He wrote, "it failed to be an adequate tool for systems programming, partly because it was burdened with feature unnecessary for systems programming tasks, and partly because it lacked adequately flexible data structuring facilities" (Wirth, 98). As a result, Wirth decided to go for his original goal of designing a general-purpose language (Wirth, 98).

Further more, Wirth found that the task of teaching programming language, such as systems programming, was very unattractive, given the choice between FORTRAN and assembler code that were available that time. Wirth thought that it was right time for structured programming, and make them applicable in practice by providing a language and compilers offering appropriate constructs (Wirth, 99). He felt that the discipline (programming) was to be taught at the level of introductory programming courses, rather than at the advanced level. Therefore, in 1968 he started to design with the two goals in his mind, " the language was to be suitable for expressing the fundamental constructs known at the time in a concise and logical way, and its implementation was to be effiecient and competitive with the existing FORTRAN compiler" (Wirth, 99).

The task of writing the compiler was assigned to E. Marmier, a graduate student, in 1969 (Writh, 99). Marmier's wrote compiler with FORTRAN, with the translation into Pascal and self-compilation planned after its completion. According to Wirth, "It was a grave mistake" (Wirth, 99). As a result, the second attempt to build a compiler began with the source language itself. N. Wirth decided that the compiler would be

single-pass system based on the top-down, recursive-descent principle for syntax analysis. In mid-1970, the compiler was completed. This compiler remained stable thereafter, except little revision in 1972 (Wirth, 100). It is important to mention here that in order to assist in the teaching effort, Kathy Jensen wrote a tutorial text explaining the primary programming concept of Pascal with many examples (Wirth, 100).

The language:

A Pascal program has three basic parts: (a) Program Heading: defines program name and program parameters, (b) Declaration Part: defines names for constants and variables, and (c) Statement Part: describes actions to be performed by the program (Horn, 42). For example:

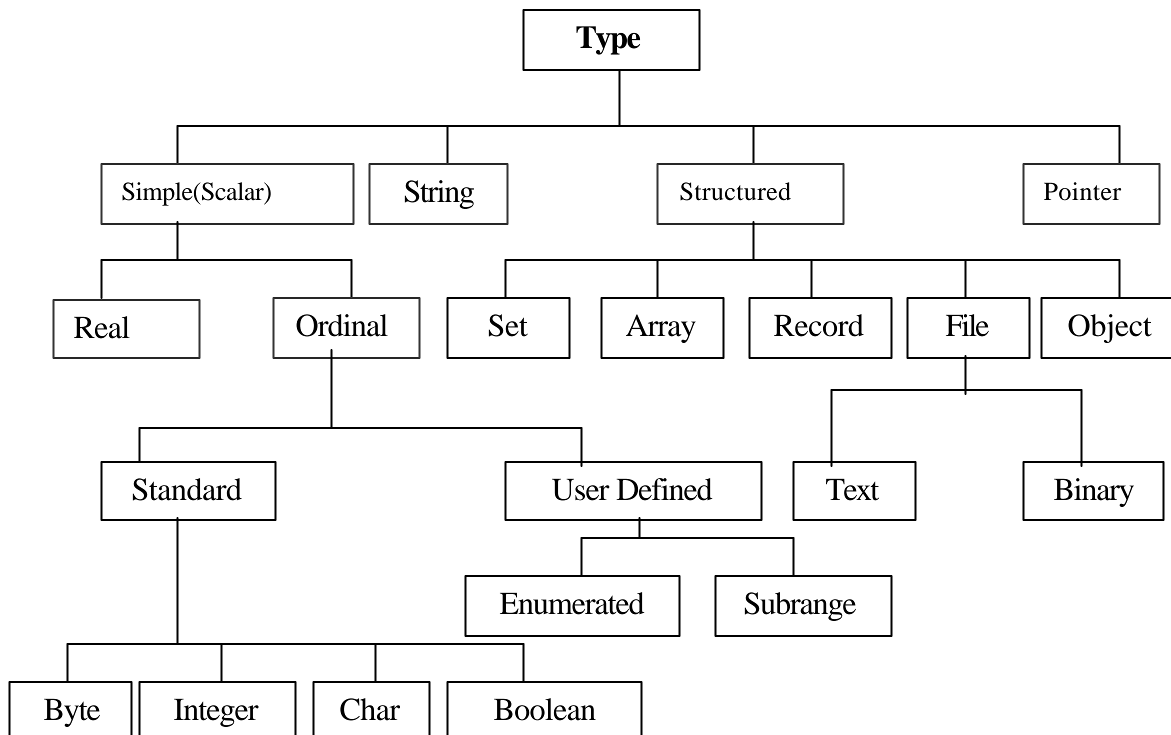
```
Program Sample {Input, Output};      {Program Heading}
Var                                  {Declaration Part}
    Name : string[20];
    Age : integer;
Begin                                {Statement Part}
    Writeln {'What is your name?');
    Readln (Name);
    Writeln {'What is your age?');
    Readln (Age);
    Writeln;
```

Writeln {'Your name is', Name, 'and you are', Age, 'years old.');

End.

In this example shows three basic parts of the program. The first line is the program heading, the second , third, and forth lines are the declaration part, and the rest of the lines are the statement part. Program- heading shows that it takes user input and show or write the out put. Declaration part declares variable or constants that assign memory space for that variables or constants. Statement part actually do the input, output and processing tasks.

It is important we discuss the Pascal data type because later we are going to use these data type to compare with other programming languages. The language supports four basic kinds of data types: (a) Simple, (b) String, (c) Structured, and (d) Pointer (Horn, 332).



Simple data types are the building blocks for other data types. The simple data types are: Integer, Real, Char, Boolean. The simple data types are the scalar, which is based on the word scale, and borrowed from mathematics. A scale is a way of ordering values. Perhaps the most common scale is the number line. Each real number can be represented on the number line by a point. For example, if real number a is less than real number b, then the point representing a lies to the left of the point representing b. Although all the simple data types are scalar, not all are ordinal. Ordinal means that objects are ordered in such a way that it is possible to determine which one is the first, which one is the second and so forth. In the context of Pascal, the term ordinal implies that it is possible to determine the successor and the predecessor of any value. All the standard scalar types except Real are classed as ordinal (Horn, 332). A two way to classify the ordinal data types is either standard or user defined. We can divide user defined data types into two parts: Enumerated and Subrange. Subrange is based on one of the ordinal data types and its expressed by writing the beginning value, two periods, and the ending value. For example, 1..10 is a subrange based on type integer. Another user defined data types are the enumerated types that users can list explicitly all instances of the data.

Another large category of types is the structured types. The term structured means that each of these types is based on an organized collection of data items of other types. For example, a set is a group of data items of a specified ordinal type such as integer or char. Record is a grouping of data of various types that can be processed as a single unit. A file is a grouping of data stored externally to the program (usually on disk). Objects are the collections of data items of various types, together with procedures and functions that process the data (Horn 333).

Pascal combined many of the best features of the languages in use at that time, such as COBOL, FORTRAN, and ALGOL. In this way, many of the irregularities and oddball statements of these languages were cleaned up in Pascal. For example, it incorporated a variety of data types and data structures. ALGOL, the predecessor of Pascal, had three basic data types which are: integer, real numbers, and truth-values, and array structure (Wirth, 101). N. Wirth wrote, "Pascal introduced additional basic types and the possibility to define new basic types (enumerations, subrange), as well as new forms of structuring: record, set, and file (sequence), several of which had been present in COBOL. Most important was of course the recursively of structural definitions and the consequent possibility to combine and nest structure freely" (Wirth, 101).

Pascal has its own innovation in the programming language. For example, it replaced ALGOL's *for*-statement with while statement. Wirth wrote, "which is efficiently implementable, restricting the control variable to be a simple variable and the limit to be evaluated only once instead of before each repetition. As a result it was impossible to formulate misleading, non-terminating statements (Writh, 101). Following is N. Wirth's example:

> "For I : = 0 step 1 until I do S
>
> And the rather obscure formulation
>
> For I : = n – 1, I – 1 while I > 0 do S
>
> Could be expressed more clearly by
>
> I : = n;

While I > 0 Do Begin I : = I – 1; S End "(Writh, 101).

Pascal also improved the "pointer" data type, a very powerful feature of any language that implements it. "The introduction of explicit pointers, that is, variables of pointer type, was the key to a significant widening of the scope of application. Using pointers, dynamic data structures can be built, as in list-processing languages. It is remarkable that the flexibility in data structuring was made possible without sacrificing strict static type checking. This was due to the concept of pointer binding, that is, of declaring each pointer type as being bound to the type of the referenced objects" (Wirth, 101). He (N. Wirth) gave the following example and explanation:

"Type pt = ↑ Rec;

Rec = Record x, y: Real End;

Var p, q: pt;

Then p and q provided they had been properly initialized, are guaranteed to hold either values referring to a record of type Rec, or the constant NIL. A statement of the form

p ↑ .x : = p ↑ .y + q ↑ .x

Turns out to be as type-safe x : = x + y" (Wirth, 101)

Another important feature of Pascal is that it permits the definition of arrays of records, records of arrays, arrays of sets, and arrays of records with files, etc. According to N. With, "naturally, implementations would have to impose certain limits as to the depth of nesting due to finite resources, and certain combinations, such as a file of files, might not be accepted at all. This case may serve as an example of the distinction

between the general concepts defined by the language, and supplementary, restrictive rules governing specific implementations" (Wirth, 103).

It also added a CASE statement that allowed instructions to branch like a tree in such a manner:

```
CASE expression OF
    possible-expression-value-1:
        statements to execute...
    possible-expression-value-2:
        statements to execute...
END (Ferguson, 2)
```

Another important feature of Pascal is the data transfer. According, "Pascal separated the notions of data transfer (to or from an external medium) and of representation conversion (binary to decimal and vice versa)… Representation conversion was expressed by special read and write statements that have the appearance of procedures but allowed a variable number of parameters. … The consequence was that, in contrast to all other data types, files require a certain amount of support from built-in run-time routines, mechanisms not explicitly visible from the program text" (Wirth, 104).

Later Development:

In 1973 another compiler called as the P-Compiler was introduced. This compiler was a part of the P-kit consisted of the compiler in P-code and the interpreter as a

Pascal's source program. The users of the P-kit could code the interpreter in assembler code, or to modify the source of the P-compiler and replace its code-generating routines. According to Wirth, "this P-system turned out to be the key to Pascal's spread onto many computers." K. Bowles and a team at the University of California at San Diego (UCSD) made the Pascal compiler to fit into the memory of a microcomputer. Moreover, the idea of P-code made it easier to port Pascal to all micro-computers and provided a common basis for teaching programming language. According to Wirth, "Bowles not only ported the compiler, his team built an entire system around the compiler, including a program editor, a file system, and a debugger. As a result it reduced the time needed for an edit-compile-test step dramatically over any other system in educational use" (Wirth, 106).

In Wirth's own analysis of P-system, he wrote, "besides being the major agent for the spread of Pascal implementation, the P-system was significant in demonstrating how comprehensible, portable, and reliable a compiler and system program could be made. Many programmers at that time learned a great deal from the P-system, including implemetors who did not base their work on the P-system, and other who had never before been able to study a compiler in details. The fact that a compiler was available in source form caused the p-system to become influential vehicle of extracurricular education" (Wirth, 106).

Pascal became popular for classroom used and for smaller software projects by 1973. One of the main reason for the users acceptance was the availability of a user manual including tutorial material in addition to the language definition. Wirth wrote, "increasing numbers of textbook had been produced from different authors in different

countries in different languages. The Users Manual itself was later to be translated into many different languages, and it became a bestseller" (Wirth, 107).

To set a standard for Pascal was important. A committee was formed to define a standard in 1977. Wirth wrote, "at the Southampton conference on Pascal, A.M. Addyman asked for help in forming a standards committee under the British Standard Institute (BSI). In 1978, representatives from industry met at a conference in San Diego hosted by K. Bowles to define a number of extensions to Pascal. This hastened the formation of standards committee under the wings of IEEE and ANSI/X3. The formation of a Working Group with ISO followed in late 1979, and finally the IEEE and ANSI/X3 committees were merged into single Joint Pascal Committee" (Wirth, 107). However, there were significant conflicts between US committee and the British and ISO. "The issue of dynamic arrays eventually led to a difference between the standards adopted by ANSI on one hand, and BSI and ISO on the other. The unexpended standard was adopted by IEEE in 1981 and by ANSI in 1982" (Wirth, 107-108). While the standard was being set, several companies had implemented Pascal and added their own extension. The effort to bring them under a single umbrella of standard was failed because many companies had produced and distributed compilers and were not willing to modify them in order to compile with the late standard (Wirth, 108).

Language Critique:

Pascal is a strongly typed language which means each object in a program has a well-defined type. Which implicitly defines the legal values of and operations on the

object (Kernighan 5). Pascal language prohibits illegal values and operations, by some mixture of compile- and run-time checking. The compilers may not actually do all the checking that implied in the language definition (Kernighan 5). For example, “if one defines types 'apple' and 'orange' with

```
Type
        apple = integer;
        orange = integer;
```

then any arbitrary arithmetic expression involving apples and oranges is perfectly legal” (Kernighan 5). In the language the strong typing shows up in many different of ways. For instance, arguments to functions and procedures are checked for proper type matching. Kernighan wrote, “as in the FORTRAN freedom to pass a floating-point number into a subroutine that expects an integer; this desirable attribute of Pascal, since it warns of a construction that will certainly cause an error” (Kernighan 5).

Some times integer variables may be declared to have an associated range of legal values, and the compiler and run-time support ensure that one does not put large integers into variables that only hold small ones. This too seems like a service, although of course run-time checking does exact a penalty (Kernighan 6). Another problem with Pascal is that the size of an array is part of its type. Kernighan gave the following example that if one declares:

```
var   arr10 : array [1..10] of integer;
        arr20 : array [1..20] of integer;
```

then arr10 and arr20 are arrays of 10 and 20 integers respectively. Suppose we want to write a procedure 'sort' to sort an integer array. Because arr10 and arr20 have different types, it is not possible to write a single procedure that will sort them both (Kernighan 6). "The place where this affects Software Tools particularly, and I think programs in general, is that it makes it difficult indeed to create a library of routines for doing common, general-purpose operations like sorting" (Kernighan 6).

According to Kernighan, "the particular data type most often affected is 'array of char', for in Pascal a string is an array of characters. For example, writing a function 'index(s,c)' that will return the position in the string s where the character c first occurs, or zero if it does not" (Kernighan 6). Now the problem is how to handle the string argument of 'index'. "The calls 'index('hello',c)' and 'index('goodbye',c)' cannot both be legal, since the strings have different lengths", Kernighan wrote (Kernighan 6). He also gave the example:

```
" var     temp : array [1..10] of char;
  temp := 'hello';
  n := index(temp,c);
```

the assignment to 'temp' is illegal because 'hello' and 'temp' are of different lengths. The only escape from this infinite regress is to define a family of routines with a member for each possible string size, or to make all strings (including constant strings like 'define') of the same length" (Kernighan 6).

However, many commercial Pascal compilers which provide a 'string' data type that avoids the problem that 'string's are all taken to be the same type regardless of size. According to Kernighan, “this solves the problem for this single data type, but no other. It also fails to solve secondary problems like computing the length of a constant string; another built-in function is the usual solution” (Kernighan 6). Other argued that to cope with the array-size problem one merely has to copy some library routine and fill in the parameters for the program at hand (Kernighan 6). However, this argument seems weak that since the bounds of an array are part of its type, it is impossible to define a procedure or function, which applies to arrays with differing bounds. “Although this restriction may appear to be a severe one, the experiences we have had with Pascal tend to show that it tends to occur very infrequently. However, the need to bind the size of parametric arrays is a serious defect in connection with the use of program libraries” (Kernighan 6).

Another problem with Pascal is that there are no static variables and no initialization (Kernighan 6). Kernighan wrote, a 'static' variable is one that is private to some routine and retains its value from one call of the routine to the next. Pascal has no storage class that means that if a Pascal function or procedure needs to remember a value from one call to another, the variable used must be external to the function or procedure (Kernighan 7). He wrote that the variables must be visible to other procedures, and its name must be unique in the larger scope. He also gave an example of the problem is a random number generator: the value used to compute the current output must be saved to compute the next one, so it must be stored in a variable whose lifetime includes all calls of the random number generator. In practice, this is typically the outermost block of the

program. Thus the declaration of such a variable is far removed from the place where it is actually used (Kernighan 7).

One example that Kernighan gave that the variable 'dir' controls the direction from which excess blanks are inserted during line justification, to obtain left and right alternately. In Pascal, the code looks like this:

```
program formatter (...);
var
        dir : 0..1;    { direction to add extra spaces }
        …
procedure justify (...);
begin
        dir := 1 - dir; { opposite direction from last time }
        ...
end;
        ...
begin { main routine of formatter }
        dir := 0;
        ...
end;
```

In this example, we see that the declaration, initialization and use of the variable 'dir' are scattered all over the program (Kernighan 7). “In C or Fortran, 'dir' can be made private to the only routine that needs to know about it:

```
main()
```

```
{
        ...
}
        ...
justify()
{
        static int dir = 0;
        dir = 1 - dir;
        ...
}" (Kernighan 7).
```

Furthermore, the lack of initializers is another problem caused by the lack of a static storage class. Kernighan wrote, "the time to initialize things is at the beginning, so either the main routine itself begins with a lot of initialization code, or it calls one or more routines to do the initializations. In either case, variables to be initialized must be visible, which means in effect at the highest level of the hierarchy. The result is that any variable that is to be initialized has global scope (Kernighan 8).

Another problem with Pascal is that related program components must be kept separate. The language believes strongly in declaration before use because the original Pascal was implemented with a one-pass compiler. For an example that procedures and functions must be declared (body and all) before they are used. The result is that a typical Pascal program reads from the bottom up - all the procedures and functions are

displayed before any of the code that calls them, at all levels. This is essentially opposite to the order in which the functions are designed and used (Kernighan 8).

There is also a 'forward' declaration in Pascal that permits separating the declaration of the function or procedure header from the body; it is intended for defining mutually recursive procedures. When the body is declared later on, the header on that declaration may contain only the function name, and must not repeat the information from the first instance (Kernighan 8). A related problem is that Pascal has a strict order in which it is willing to accept declarations. Each procedure or function consists of :

label label declarations (if any)

const constant declarations (if any)

type type declarations (if any)

var variable declarations (if any)

procedure and *function* declarations (if any)

begin

body of function or procedure

end

According to Kernighan that this means that all declarations of one kind (types, for instance) must be grouped together for the convenience of the compiler. Since a program has to be presented to the compiler all at once, it is rarely possible to keep the declaration, initialization and use of types and variables close together (Kernighan 9). “The inability to make such groupings in structuring large programs is one of Pascal's most frustrating limitations” (Welsh, 688).

Another problem is that there is no separate compilation for the “official” Pascal language does not provide separate compilation. As a result the implementation decides on its own what to do (Kernighan 9). According to Kernighan “some (the Berkeley interpreter, for instance) disallow it entirely; this is closest to the spirit of the language and matches the letter exactly” (Kernighan 9). However, many others provide a declaration that specifies the body of a function is externally defined. In this case, all these mechanisms are non-standard, and thus done differently by different systems (Kernighan). Kernighan wrote, “however, theoretically, there is no need for separate compilation - if one's compiler is very fast (and if the source for all routines is always available and if one's compiler has a file inclusion facility so that multiple copies of source are not needed), recompiling everything is equivalent. In practice, of course, compilers are never fast enough and source is often hidden and file inclusion is not part of the language, so changes are time-consuming” (Kernighan 8).

Additionally, there are significant problems with control-flow and source code organization. There is "no guaranteed order of evaluation of the logical operators and and or" (Kernighan 7). This disallows tests like "while (i <= XMAX) and (x[i] > 0) do ..." because the programmer can't be assured that the left test will be evaluated before the right test. There is also no break statement for exiting loops and no return statement for functions. This is a result of the one in-one out design of Pascal, which can be a useful restriction in terms of source code analysis, but it forces the programmer to write unnecessarily confusing code in some cases. The fact that there is no default clause in

cast statements makes the lack of a break more cumbersome, and generally makes the case construct unusable (Kernighan 7 - 9).

Conclusion:

In conclusion, Pascal extended a strong influence on the field of language design. It acted as a catalyst for new ideas and as a vehicle to experiment with them, and in this capacity gave rise to several successor languages (Wirth, 108). For example, P. Brinch Hansen's Concurrent Pascal, Pascal-Plus developed by J. Welsh and J. Elder at Belfast. Pascal was the ancestor of the language Mesa which had added a revolutionary concept of modules with import and export relationships, that is, of information hiding. Another derivative of Pascal is the language Euclid. Object Pascal was another extension of Pascal incorporating the notion of object-oriented programming, that is, of the abstract data type binding data and operators together. The language Ada, that I mentioned before, was influenced by Pascal. Therefore, Pascal was forerunner for many of the languages. It is worth reading the history of Pascal.

BIBLIOGRAPHY:

Bergin, Thomas J. and Richard G. Gibson, eds. *History of Programming Languages-II*. New York: ACM Press, 1996.

Wirth, N. “Recollections About the Development of Pascal.” *History of Programming Languages-II*. New York: ACM Press, 1996.

Kernighan, Brian W. "Why Pascal is Not My Favorite Programming Language." Murray Hill, New Jersey: AT&T Bell Laboratories, 1981.

Horn, Wayne L. “Structure Programming in Turbo Pascal” 2nd ed. Prentice Hall, Englewood Cliffs, New Jersey: 1995.

Clippinger, R. F. “ALGOL – A Simple Explanation” *Computers and Automation*. February, 1962.

J. Welsh, W. J. Sneeringer, and C. A. R. Hoare. "Ambiguities and Insecurities in Pascal." *Software Practice and Experience 7*, pp. 685-696 (1977).

Deitel, H. M. & Deitel, P. J. “Java How to Program” 3rd ed. Prentice Hall, Upper Saddle River, New Jersey: 1999.

Williams, M. R. “History of Computing Technology” 2nd ed. IEEE Computer Society Press, Los Alamitos, CA: 2000.

Bergin, Thomas J. History of Computing Class Handout: February 01, 2001.

Ferguson, Andrew. "The History of Computer Programming Languages." http://www.princeton.edu/~ferguson/adw/programming_languages.shtml. August 03, 2000.