PASCAL-F : A PORTABLE FORTRAN-BASED PASCAL COMPILER

by

JOSEPH MANNING

B.Sc., The National University of Ireland, 1974

M.Sc., The National University of Ireland, 1975

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

September 1981

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date SEP 14, 1981

ABSTRACT

This thesis examines the subject of compiler portability and describes a project which adopts one particular approach to achieving this goal: generate code in an existing widely-implemented language as output from the compiler. Pascal-F translates Pascal to F-code, an intermediate language which can be processed directly by any Standard FORTRAN compiler. The design of the F-code machine is treated in detail, and problems with the use of FORTRAN as a target language are discussed.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to express my sincere thanks to my Supervisor, Professor Harvey Abramson, for his constant encouragement, optimism, guidance, and patience, during the development of Pascal-F and the writing of this thesis.

This thesis is dedicated

to all the friends I have known

in beautiful Vancouver

Chapter 1 : INTRODUCTION

With the ever-expanding use and application of computers, the topic of Software Portability has become increasingly important. Portability is a measure of the ease with which a particular piece of software may be transferred from one environment to another, relative to its initial implementation effort. Such a transfer could be either spatial (between two computer installations) or temporal (between an old environment at an installation and a newer replacement), and the change of environment could be to a different machine, or operating system, or both. Portable software offers a number of considerable benefits:

- the cost of implementation in a new environment is greatly reduced if the software can be transferred easily from an existing implementation, thus avoiding either a total rewrite or a major conversion effort (the current annual cost of transferring software is estimated to be in excess of $10 billion in the U.S.A. alone[3], much of which would be saved if the original software were written in a more portable manner).
- if the transfer process is simple so that the likelihood of introducing new errors is small, then the transferred software will generally be far more reliable than a separately-written version since it will have been tested by use in its original environment.

- in the academic and research worlds, portability of software encourages communication, co-operation, software interchange and unity of research, while avoiding much duplication of effort.

In the early days of computing, only machine and assembly languages were available for the writing of software. Portability was then nearly impossible to achieve, since all software written in these low-level languages had to be specifically oriented towards the details of its environment. The situation was greatly improved with the advent and widespread use of high-level languages, which were designed to be uniform across all environments. Software written in one of these languages could be transferred to any environment in which the language was implemented. (In practice, however, problems still remained due to non-uniform implementations: the lack of rigidly-defined language standards gave rise to differing interpretations of certain features; local "bells and whistles" were added as extensions; environment-dependencies, such as details of the character set or machine wordlength, were often still present in programs; etc. See Poole[13] for a lengthy discussion).

However, in order to make a given high-level language widely available, some form of compiler or interpreter must be provided for each environment. The most common approach in the past was to write a separate compiler, generally in assembly

language, for each new and substantially different environment, with very little emphasis on portability. More recently, much research has been done and considerable progress made in producing compilers which are portable. These provide all of the benefits outlined above, with increased reliability being of particular importance in compilers. In addition, the use of portable compilers leads to far more standardized language implementations, thereby contributing to the portability of software in general.

Since a compiler must ultimately generate machine language which itself must interact with its operating system, it is clear that compilers, by their very nature, are far more environment-dependent than general applications software. As a result, in the overall study of software portability the topic of compiler portability is a particularly interesting and challenging one.

This thesis describes the design and implementation of Pascal-F, a highly portable compiler for the programming language Pascal[7,6]. The Pascal-F compiler translates Pascal source programs into F-code, an intermediate language which may be processed <u>directly</u> by any Standard FORTRAN compiler to generate machine language. Chapter 2 discusses the overall question of compiler portability, studies some general solutions, and presents the Pascal-F approach. Chapter 3 presents the F-code machine and describes the various factors

which influenced its design. Chapter 4 gives a general description of the Pascal-F translator and shows how the various syntactic constructs in Pascal are translated into F-code. Chapter 5 concludes the thesis with some reflections on the project.

Chapter 2 : COMPILER PORTABILITY

A compiler is itself a computer program. Its task is to analyse programs written in a high-level language and translate them into equivalent programs in the machine language of a given computer. Thus, compiler portability has two distinct aspects:

- the ability to transfer the compiler itself to a new environment.
- the ease with which the compiler can be altered to generate machine language for this new environment.

Both of these aspects are now examined. The former relates to the general problem of software portability; the latter specifically to compiler portability.

In order that the compiler itself may be easily transferred, like any portable software it should be written in a language which is environment-independent. At first sight the most appropriate choice would seem to be an existing high-level language which has already been widely implemented. However, most portable compilers (e.g. BCPL[15], Pascal-P[12], Bliss[21], C[8]) are in fact written in the very language which they themselves compile[10] and are generally implemented at a new installation via some form of bootstrapping technique. Such self-compiling compilers have the important advantage of now being independent of the availability or correctness of another specific language at the new installation.

In dealing with the second aspect of compiler portability, experience has shown that the task of modifying a compiler to produce machine language for a new environment is considerably simplified if that part of the compiler involved in code generation is clearly separated by a well-defined interface from the remainder of the compiler. Thus it is usual for a portable compiler to be structurally divided into what Richards[14] terms a <u>syntactic phase</u> (SP) and a <u>code generator phase</u> (CGP):

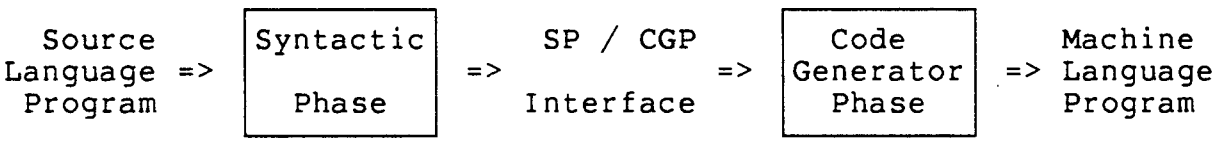

Fig. 1 : Structure of a Portable Compiler

The SP reads the source language program, breaks it into a stream of lexical tokens, and parses these to determine its syntactic structure; it will also issue diagnostics if syntax errors are encountered. Having established the structure of the entire program or of an individual construct, the SP determines to some degree what actions are required for its execution. This information is then transmitted across the interface to the CGP, which generates a corresponding sequence of instructions in the machine language of the given computer.

The SP is almost totally independent of its environment so that apart from some minimal adjustments (e.g. to details of the character set or the value of the largest integer) it can remain unchanged during a transfer of the compiler. On the other hand,

the CGP will obviously need to be altered considerably to suit different environments. Note that the presence in the source language of those features which can be handled completely by the SP, such as rich control structures, will not hinder portability, whereas features involving the CGP, such as the provision of a REAL data type, may cause difficulties.

The SP / CGP interface can assume a number of very distinct forms. It could consist of a data structure, such as a parse tree which the SP builds and the CGP subsequently traverses. Alternatively, it could simply be a set of procedure calls; once the SP has determined what semantic actions are required for the current syntactic construct, it issues a direct call to the relevant procedure in the CGP. A further possibility, which is now examined in detail, is perhaps the most flexible of all such interfaces, particularly with respect to portability.

A. The Abstract Machine Approach

The operational semantics of the source language may generally be formulated in terms of a set of primitive operations, which are independent of both the original language syntax and any particular environment. An abstract machine is a (hypothetical) computer whose instruction set consists of precisely these operations; its assembly language comprises the associated intermediate code. Having defined an abstract machine for the source language, its intermediate code can serve

as a very suitable interface between the SP and CGP of the compiler. The SP translates the source language program into a program in the intermediate code; the CGP in turn translates this into a machine language program.

Clearly, for any particular source language there are many possible designs for an abstract machine. The general design criteria will include such factors as the overall cleanliness of the resulting machine, its suitability for expressing the semantics of the source language, the compactness and efficiency of its intermediate code, the ease with which this can be generated by the SP, and the ease of implementing the CGP across a variety of environments. These criteria frequently conflict amongst themselves; the compromise reached will depend on their relative importance in a given situation.

The abstract machine approach has proved to be very successful in the construction of portable compilers for a number of languages. The most notable and well-known examples include the OCODE and INTCODE machines for BCPL[15], the P-code machine for Pascal[12] and the Z-code machine for Algol 68-C[2]. A major advantage with respect to portability of having the SP / CGP interface in the explicit form of intermediate code is that for self-compiling compilers, such as those just mentioned, the compiler itself can be distributed as a program in the intermediate code, so as soon as the CGP has been implemented at an installation the entire compiler becomes available.

The task of implementing the CGP is also simplified and very clearly defined by the explicit form of the SP / CGP interface. It basically consists of providing a translation into machine language for each of the different intermediate code instructions, which are generally very simple in nature. (An outstanding example of such simplicity is INTCODE, which is composed of essentially only 8 very elementary instructions; however, this case is somewhat extreme, and P-code, with some 58 instructions, is far more typical of the general situation). A common technique is to use a macro-processor in implementing the CGP, writing an individual macro for each intermediate code instruction to produce the corresponding machine language.

An alternate approach is to realize the CGP in the form of an interpreter for the intermediate code instructions. This method has an added advantage in that it completely avoids the problem of generating machine language, thereby resulting in even greater portability. It has been used to successfully implement languages such as SNOBOL4[5] and Pascal-S[20].

B. A Standard Abstract Machine

As outlined above, an abstract machine is designed to mirror the primitive operational semantic actions of a particular language as conveniently as possible. It is not surprising then that abstract machines for different languages are themselves quite different. Nevertheless, there is still a

remarkable degree of similarity between a lot of the primitives for many languages and this suggests the very interesting and powerful concept of a standard abstract machine (SAM), i.e. a single abstract machine capable of expressing and implementing the primitives of all languages.

Although its design poses many problems, the creation of a SAM would clearly have a tremendous impact on the whole question of compiler portability. Once a SAM became widely implemented, the task of transferring a SAM-based compiler to a new environment would now be greatly simplified since a suitable CGP (i.e. precisely the SAM's own implementation) would already be available at this new environment.

The SAM concept is not new; in fact it was suggested as far back as 1958[11]. The intermediate code for this proposed SAM was known as UNCOL (UNiversal Computer Oriented Language). The basic motivation was to simplify the problem of implementing L languages in E environments. Using the UNCOL SAM (!), only a single SP need be written for each language (translating the language to UNCOL), and similarly a single CGP for each environment (translating UNCOL to machine language). This yields a total of only L + E translators compared with the L * E translators which would otherwise be required.

A key factor in the UNCOL scheme was that to achieve portability, the SP for each language should itself be written in UNCOL. This added greatly to the task of designing UNCOL,

since it now had to be rich and expressive to facilitate the writing of these translators, as well as being simple enough both to reflect the very primitive operations of a wide variety of languages and to be easily implemented across a range of environments. These difficulties were further compounded by the fact that compiler-writing techniques at that time were not well developed. Although a certain amount of initial progress was made[17], the original UNCOL project was finally abandoned.

A more recent attempt by Waite's team at the University of Colorado, based on many years of advances in compiler-writing, has resulted in the formulation of the "universal intermediate language" Janus[4]. The results are encouraging: Janus has been used successfully in the construction of compilers for BCPL and Pascal[18], and has been implemented in a number of different environments. By contrast with the original UNCOL, Janus is intended to be used solely as an intermediate code and not as a problem-oriented language for writing the SP of a compiler.

C. The Pascal-F Approach

The SAM concept has tremendous potential as regards compiler portability. However, its realization is a monumental undertaking; not alone must the machine be designed to uniformly cater for the primitives of a whole spectrum of languages, but it must subsequently be implemented across a wide range of environments. The Pascal-F project described in this thesis

is an attempt to retain the advantages of a SAM while avoiding both of these major difficulties.

The central idea underlying the Pascal-F approach is to generate an intermediate code which can be processed directly by the compiler of an existing widely-implemented language.

This approach offers two significant advantages which solve the corresponding problems above. Firstly, the resulting somewhat unspecified abstract machine is exceptionally flexible and extensible — almost any semantic action can be realized merely by coding an appropriate target language routine and adding a new intermediate code instruction which calls this routine. Thus the abstract machine is very easily adapted to incorporate the primitives of a great variety of languages. Secondly, the problem of general implementation of the abstract machine now no longer arises — the target language's compiler serves directly as the CGP for the intermediate code. Such a compiler will already have been thoroughly tested by other uses, and can therefore be expected to be far more reliable than any newly-written special-purpose CGP.

Clearly, the success of this approach hinges on the widespread availability of the target language selected. Of all existing languages, at present the most universally implemented is probably FORTRAN. Because of this (and this alone!), it was the target language chosen for the current project. More precisely, ANSI (1966) Standard FORTRAN[1,16] was used;

although most "FORTRAN IV" compilers feature a considerable number of enticing extensions to the standard language, these are very implementation-dependent and must clearly be avoided since portability is of the utmost importance. In many ways, FORTRAN is far from ideal for this task, since its numerous restrictions as a compiler-oriented language (in particular, the absence of recursion and low-level I/O) present many difficulties. From this standpoint, a language such as BCPL or C would have been much more appropriate; however, these languages are not widely available and since this factor was of prime concern, FORTRAN was the eventual choice. The difficulties were then accepted as a challenge to be overcome in establishing the viability of the overall approach.

In keeping with the great traditional scheme for naming intermediate codes, that of the FORTRAN-based SAM is known as F-code. It is important to again note that F-code is not a fixed static language. Rather, it consists of a general framework with a certain core set of instructions (primarily dealing with routine invocation and I/O), to which new instructions may be added with ease to meet the needs of differing situations.

The Pascal-F project itself consisted of defining and creating a suitable F-code machine for the language Pascal and writing a translator from Pascal to F-code. A fairly simple stack machine was designed, implemented and found appropriate;

it is described in some detail in the following chapter. Once the machine's overall structure had been decided upon, the development of the individual F-code instructions proceeded hand-in-hand with the writing of the translator, with new instructions being added as the need arose. This served to demonstrate the inherent flexibility of the method.

Although it was at first hoped that this F-code machine could be implemented entirely using only Standard FORTRAN and so yield an almost totally portable compiler, after much effort it was eventually found that a certain small number of operations just could not be realized this way. These included operations to empty an external file, to read a variable-length input line, to detect the end-of-file condition, etc.; they are discussed in more detail later. The solution reached was to carefully isolate these operations and implement each by means of an environment-dependent routine, using either assembly language or, as was sometimes possible, extensions to FORTRAN. Although this solution unfortunately conflicts somewhat with the original design concept, nevertheless the resulting interface which must be written for each environment is very small, clearly-defined and straightforward to implement, and the overall compiler still remains extremely portable. In fact, by contrast with all of the other portable compilers that were examined (particularly those for Pascal — see Lecarme and Thomas[10]), the transfer of Pascal-F to a new environment involves only a fraction of the effort which is normally required.

The Pascal to F-code translator, i.e. the SP of the compiler, was written in Pascal. Note that this differs from the UNCOL approach which would have required it to be written directly in F-code, a much more difficult undertaking. Pascal was used because of its general suitability for such a task, the overall clarity, modifiability and reliability of the resulting translator, and of course chiefly for the implications of self-compilation on portability. Once an original version of the translator was written, it was firstly compiled under an existing Pascal compiler, and then executed with itself as input. This produced an F-code program capable of translating Pascal to F-code! The Pascal-F compiler was then completely self-contained, and the existing Pascal compiler could be discarded. This self-compilation was carried out as soon as the translator could handle all the features necessary to translate itself. Further expansion of the translator was then possible by modifying the Pascal version, and translating this under the current F-code program to yield an F-code version of the expanded translator.

Chapter 3 : THE F-CODE MACHINE

This chapter presents the underlying factors which shaped the design of the F-code machine, and describes the result which finally emerged. Although the presentation here is given in terms of planning the machine as a target specifically for the translation of Pascal, this is done primarily for illustrative purposes. It will be clear that many of the conclusions arrived at apply equally well in a more general setting.

A. The Structure of F-code Programs

One of the earliest and most fundamental decisions concerned the overall structure of the F-code program resulting from the translation of a Pascal program. The simplest and most appealing structure is the monolith — translate the Pascal program together with all its enclosed procedures and functions into one large single main program in F-code. This approach had been very successfully used in a previous course project to implement a subset of Pascal, which included full nesting of procedures and functions but was never meant to be self-compiling. Routine calls can be implemented in a highly efficient manner by simply ASSIGNing the label of the following statement to a "return address" variable and performing a GOTO the start of the routine's code. This scheme also very nicely solves a substantial problem connected with the translation of recursive routines. In fact, the monolithic approach is ideally

suited to the task, and there is no fundamental reason why it could not be used; however, it had to be rejected because of a fairly mundane consideration. The all-important F-code version of the Pascal-F translator itself is close to 10,000 lines long. A single main program of this size is far beyond the capacity of the average FORTRAN compiler — experiments showed that even at a reasonably large installation (IBM 4341), a monolith of fewer than 3,000 "typical" F-code lines already exceeds the compiler limits. Since it is essential for portability that Pascal-F itself should lie within the capacity of a wide range of FORTRAN implementations, clearly the monolithic approach is not feasible.

The alternative is to translate each procedure and function of the Pascal program into a separate SUBROUTINE in F-code. This approach was adopted and ultimately proved successful. Assuming that the Pascal source program has been suitably organized as a series of subprograms, its F-code translation will now consist of a number of short separate units which can easily be handled individually by any FORTRAN compiler. (Note that functions in Pascal become SUBROUTINEs, rather than FUNCTIONs, in F-code. This results in a far simpler and more uniform scheme while avoiding a number of troublesome details associated with FUNCTIONs in FORTRAN).

B. The Problems of Recursion

However, a major difficulty was soon encountered using this approach: recursion is allowed in Pascal but not in FORTRAN. This single factor was to have a very profound impact on the overall design of the F-code machine. It gives rise to two serious problems which are now illustrated.

Consider the following recursive Pascal procedure (which simply writes out each number from 1 to N), and an attempt to translate it directly into FORTRAN:

```
procedure COUNT (N : INTEGER);
  begin
  if N > 1 then COUNT(N-1);
  WRITELN(N)
  end
```

```
SUBROUTINE COUNT (N)
IF (N .GT. 1) CALL COUNT(N-1)
WRITE(-,-) N
RETURN
END
```

Fig. 2 : Direct Translation of a Recursive Procedure

The first problem concerns storage allocation for local variables. In the Pascal version, each incarnation of COUNT is dynamically allocated a new memory location for N. On the other hand, FORTRAN generally uses a static storage allocation scheme, since recursion need not be catered for. This means that in the FORTRAN version, each incarnation of COUNT will use the same memory location for N, and consequently on every recursive call COUNT will alter the value of its predecessor's N. (One attempt to run the program produced a series of "1"s as output).

The second problem concerns the RETURN from a routine, and stems from basically the same cause. In many implementations, each FORTRAN routine is provided with one fixed location in which to store its return address, i.e. the position in its caller's code to which it should return. Suppose a main program issues a CALL COUNT(2). COUNT stores a position in the main program as its return address, and proceeds to call itself recursively. The second incarnation of COUNT now stores a position in COUNT as its return address, thus overwriting the link back to the main program. When the first incarnation eventually tries to return, it jumps to the position in COUNT and not in the main program, and of course an endless loop results. Although certain implementations (e.g. that on the VAX-11/780) avoid this problem by providing a separate location for the return address of each call, the above scheme is used on many machines (e.g. IBM 4341, Amdahl 470); clearly the F-code machine must be designed to work in all situations.

There exists a fairly standard technique for simulating recursion in FORTRAN, which is well presented in Larmouth's paper[9]. Each local variable in a recursive routine is replaced by a pushdown stack, with a further such stack being used for indicating statement labels to which control is transferred upon "returning". The resulting routine is never actually called recursively; instead, it is called once and simply jumps around within itself, pushing and popping local variables and statement labels, until its task is accomplished.

The transformed routine is written entirely in Standard FORTRAN. The method appears to be applicable to all directly recursive routines, and the required transformations should not be very difficult to automate. However, it has a number of serious drawbacks which led to its rejection as a technique for F-code: the resulting routines are somewhat lengthy, the introduction of numerous individual stacks consumes a lot of storage as well as creating dangers of stack overflow at many points, and worst of all, the method becomes extremely complicated when provision is made for mutually recursive routines.

A far simpler and more elegant solution was devised for the F-code machine and this is now described. Storage allocation is dealt with first; later it is shown how the problem with return addresses was solved.

C. Storage Allocation and Addressing of Variables

The F-code machine has one single stack, which is used for implementing all variables in all routines of the translated Pascal program. Thus, while procedures and functions in Pascal become SUBROUTINEs in F-code, there is no correspondence between their local variables — in fact, F-code SUBROUTINEs have no variables whatsoever! (They do have four special-purpose state indicators, but these are rarely used and do not really behave as variables). It is of interest to note that the F-code itself never actually provides a declaration of this stack, and has

little knowledge of its precise form or implementation; it may be regarded as part of the "hardware" of the F-code machine, manipulated only by means of particular instructions.

The stack is ultimately realized as a large one-dimensional array of INTEGERs. Pascal's fundamental data types (integer, char, boolean, enumerated types) may all be easily mapped to INTEGERs using their ORD values; Pascal's data type real is catered for by the very convenient device of EQUIVALENCEing an array of REALs over the original stack, so that stack elements may also be used to hold real data directly.

The stack is organized in a conventional manner, similar to that found in many compilers for block-structured languages. Whenever a routine is entered, storage is allocated on the top of the stack for its local variables (as well as certain linkage information); this storage is later released when the routine exits. A major advantage of this scheme is its economy of storage, since the same area can be reused for a number of routines at different times.

The bottommost stack position allotted to a routine is known as its <u>base</u>. Within a routine, each local variable is assigned a fixed <u>offset</u>, denoting its relative position in the routine's data area, with offsets being assigned to variables in the order in which they are declared. The run-time <u>address</u> of a variable, i.e. its position on the stack, is then given by:

ADDRESS of variable = BASE of routine + OFFSET of variable

Note that each variable's offset is a fixed constant which can be determined at compile-time, whereas the value of a routine's base depends on which routines are currently below it on the stack, and so cannot be known until run-time.

Because of Pascal's nested scoping rules, when any routine is executing it can access the variables of all textually enclosing routines (which clearly are already active on the stack). Thus, for the addressing of non-local variables, the base values for all such routines must be available. A special array BASE — often known as a "display" — is therefore maintained, where each element BASE(k) stores the base of the (unique) routine with static nesting level k which textually encloses the currently executing routine. The static level of each routine is of course known at compile-time. This serves as an index into BASE, and so the above formula may be used in the addressing of variables at all levels. (For the present discussion, only simple variables are considered; structured variables such as ARRAYs and RECORDs involve some extra details and will be dealt with later).

Clearly, the display BASE must be updated when a routine is entered, and restored to its former value when the routine exits. Suppose a level m routine Rm calls a level n routine Rn. Since this implies that the name Rn must be known to Rm, exactly

one of the following three cases must hold:

- Rn encloses Rm (n < m)
- Rn and Rm are at the same level and are directly enclosed by the same routine (n = m)
- Rn is directly enclosed by Rm (n = m+1)

In all cases, Rn and Rm are both textually enclosed by the <u>same</u> level n-1, n-2, ... 2, 1 routines, so that the first n-1 entries in BASE remain unchanged when Rm calls Rn and only the single entry in BASE(n) needs to be altered. Hence, the entire manipulation of the display reduces to:

CALL ROUTINE Rn : Save the current value of BASE(n) and set BASE(n) to the newly-allocated base of Rn.

RETURN FROM Rn : Restore BASE(n) to its previously-saved value.

(The old BASE(n) value is stored on the stack in the "linkage information" area of Rn). Note that this scheme is both simpler and more efficient than the widely-used method, described by Wirth[19], where an explicit "static chain" is maintained for updating the display.

There is one situation however which cannot be handled by the above scheme and must be treated specially: call-by-routine. If Rm takes a formal routine parameter Rf and is thereby passed Rn, then Rm need no longer know the name of Rn directly and so the static environments of these two routines can differ

drastically. To cater for this, a special display-sized region is reserved in Rm's data area. When Rm is invoked with actual parameter Rn, the static environment of Rn is calculated and copied into this special display. When Rm now calls Rf (= Rn), the current and special displays are swapped, thus establishing the correct static environment for Rn. When Rn returns, the displays are swapped back. Note that while Rn is executing, it accesses the display in the normal way — it does not even know that it is a parameter! The extra work outlined above will not seriously degrade overall performance since in general routines are very rarely passed as parameters.

All F-code SUBROUTINEs are parameterless, since parameters behave as local variables and hence encounter the aforementioned problem in cases of recursion. The parameters of a Pascal routine are instead allocated storage on the stack, directly after the "linkage information" of its corresponding F-code SUBROUTINE. The three different parameter-passing mechanisms allowed in Pascal are handled as follows:

CALL BY VALUE : the actual parameter must be an expression; it is evaluated and its value loaded onto the stack.

CALL BY ADDRESS : the actual parameter must be a variable; at the time of call, its address can be completely determined, and this address is loaded onto the stack.

CALL BY ROUTINE : the actual parameter must be a routine; its "routine number" (see section D below) is loaded onto the stack.

The parameters are loaded into the data area of the called routine. Note that this loading is done by the caller, before the actual transfer of control takes place.

The value of a Pascal function may be simply treated as another call-by-address parameter, with the address in this case referring to an anonymous location in the caller's data area to which the function value is returned.

All storage allocated to a routine is released upon exit, by resetting the stack pointer TOP to below the routine's base. To facilitate this, a special variable BASEXC points to the base of the executing routine, which in turn points back to the base of its caller — this is the standard "dynamic link". The top portion of the stack when a routine is entered is shown below; note that the bottom three locations are used for the linkage information, so that the offset values begin at 3.

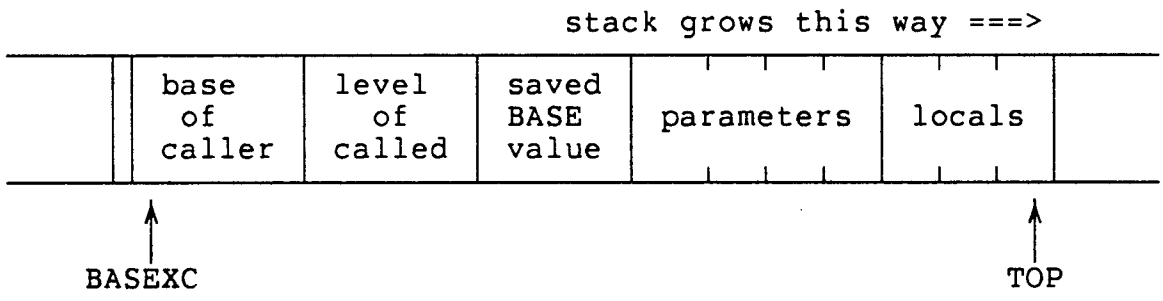

Fig. 3 : Storage Allocation for a Routine

D. The Routine-Calling Mechanism

The problem concerning return addresses can now be solved by storing them on the stack. Before calling a routine, the caller firstly allocates a "save area" on the stack, immediately below the base of the called routine, for storing the address in its code to which control should later be returned. Upon exit from the called routine, this return address is retrieved from the stack and used in a direct jump back to the caller.

The actual storing of the return address and the subsequent return jump are performed by the routines SAVENV and GOBACK respectively. These routines are highly machine-dependent and must generally be written in assembly language. (However, they are typically very simple; for example, on the IBM 4341 or Amdahl 470, they have a combined total of essentially only 11 instructions). They form part of the small interface that must be written to install the F-code machine in each environment. Fortunately, in some cases (e.g. VAX-11/780) return addresses are automatically saved on the machine's own stack, so that SAVENV and GOBACK are then unnecessary and can simply be empty routines.

For certain FORTRAN implementations, it was found that some additional registers from the caller also have to be preserved during a routine call. In such situations, the save area can be enlarged, and SAVENV and GOBACK used to save and restore all of the necessary environment.

A special routine CALLUP acts as an interface between the caller and the called routines. It is listed overleaf, and summarizes the actions involved in routine-calling. Control is transferred between routines according to the scheme:

caller ===> CALLUP ===> called ===> CALLUP ===> caller

The return "called ===> CALLUP" is simply carried out by the RETURN statement in the called routine. However, (unless the machine itself stacks return addresses), the final return "CALLUP ===> caller" involves the stacked return address and will actually be performed by the routine GOBACK, not by CALLUP's RETURN statement, since the single location provided for CALLUP's return address would be overwritten in the case of nested calls.

Each Pascal routine is assigned a unique <u>routine number</u>, and F-code routines are correspondingly named R1, R2, R3, etc. (For uniformity, Pascal's main program is also treated as a routine — R1 — which is automatically invoked from the F-code main program). This number serves as the first parameter to CALLUP and identifies the routine to be called. It also provides an extremely convenient method of dealing with call-by-routine parameters: as mentioned earlier, the routine number is passed on the stack and can later be used directly, as CALLUP's first parameter, to invoke the passed routine.

```
      SUBROUTINE CALLUP (NUMBER, LEVEL, PARSIZ)
C
C >>> Routine Calling Interface for F-code
C
      INTEGER NUMBER, LEVEL, PARSIZ, SAVEPT, OLDBSX, LEV
      INTEGER STACK(5000), TOP, BASE(30), SAVESZ, LINKSZ, BASEXC
      COMMON  STACK, TOP, BASE, SAVESZ, LINKSZ, BASEXC
C
C >>> Save the caller's environment on the stack
C
      SAVEPT = TOP - PARSIZ - LINKSZ + 1
      CALL SAVENV(STACK(SAVEPT))
C
C >>> Set the dynamic link  and  update the display
C
      OLDBSX = BASEXC
      BASEXC = SAVEPT + SAVESZ
      STACK(BASEXC) = OLDBSX
      STACK(BASEXC + 1) = LEVEL
      STACK(BASEXC + 2) = BASE(LEVEL)
      BASE(LEVEL) = BASEXC
C
C >>> Call the routine
C
      GOTO (1, 2, ..., n) NUMBER
    1 CALL R1
      GOTO n+1
    2 CALL R2
      GOTO n+1
    - - - - - -
    - - - - - -
    n CALL Rn
      GOTO n+1
C
C >>> Reset the display  and  release storage
C
  n+1 LEV = STACK(BASEXC + 1)
      BASE(LEV) = STACK(BASEXC + 2)
      SAVEPT = BASEXC - SAVESZ
      BASEXC = STACK(BASEXC)
      TOP = SAVEPT - 1
C
C >>> Restore the caller's environment  and  return
C
      CALL GOBACK(STACK(SAVEPT))
      RETURN
      END
```

Fig. 4 : The Routine CALLUP

Notes on CALLUP:

- NUMBER = routine number of called routine

  LEVEL = static nesting level of called routine

  PARSIZ = size of parameter block of called routine

  SAVESZ = size of save area

  LINKSZ = SAVESZ + 3 (for linkage information)

  n = total number of routines in Pascal source program.
- Since n depends on the particular program being translated, the routine CALLUP must be written each time by the translator.
- Before invoking CALLUP, the caller will already have created space on the stack for both the save area and the linkage information. This must be done by the caller, since it now proceeds to load the actual parameters onto the stack beyond this space.
- A full example of the entire routine-calling mechanism may be found in Appendix B (Towers of Hanoi).

E. F-code Instructions and The Run-time Library

The Pascal-F translator takes a Pascal program and from it generates an equivalent F-code program. The overall structure of this resultant program has already been described; its individual instructions are now considered.

It must first be emphasized that every instruction in F-code is a FORTRAN statement — F-code is not in any way

"translated" into FORTRAN, F-code actually is FORTRAN! Nevertheless, for the general discussion it still remains conceptually simpler and more convenient to view F-code as the assembly language of an abstract machine which has a direct "implementation" in FORTRAN.

Some F-code instructions are simple FORTRAN statements, but the majority require more complex actions and take the form of a call to a corresponding routine. Collectively, these routines constitute the run-time library. A complete listing of this library is given in Appendix C, together with a full explanation of each associated F-code instruction. Note that this provides a total definition of the F-code machine for Pascal.

The run-time library may be divided into the following major sections (for details, consult the listing):

ENVIRONMENT INTERFACE:

Four routines interface the F-code machine with the real computer; they provide F-code with details of the character set and algorithms for packing / unpacking individual words. A further four routines are used to interface with the operating system; these deal with certain I/O primitives that either cannot be implemented in Standard FORTRAN or will vary between different systems. The particular interface given in the listing is that for the Amdahl 470 running under MTS. Strictly speaking, these routines do not correspond directly to any F-code instructions, but serve to isolate all of the

environment-dependent actions needed by the other routines. (The interface routines SAVENV and GOBACK are generally written in assembly language and are not included in the listing. They are invoked only by CALLUP).

STACK MANIPULATION:

The Pascal-F translator generates postfix code for expressions and this is executed on the stack in the conventional manner. The corresponding stack instructions include various loads and stores, together with a wide range of arithmetic, logical, set-theoretic and relational operators. It may be of interest to note that after this set of instructions had been fully developed, it was discovered that many have almost exact counterparts in OCODE, the intermediate code of BCPL. This concurrence, while not totally surprising, does provide a little extra encouragement for the feasibility of a standard abstract machine.

INPUT / OUTPUT:

FORTRAN's READ and WRITE statements always transfer a complete record each time they are executed. Pascal's more flexible READ and WRITE, on the other hand, can successively move along on the same record. To bridge the gap between these two methods, the F-code run-time library explicitly implements a full buffered I/O system. Associated with each file is a buffer LINE, which stores (in integer form) the characters of the line currently being read or written. For input, a line

is read and placed in the buffer by the special system interface routine GETLN, which also returns the length of the line and an end-of-file flag (these cannot be determined using Standard FORTRAN). For output, the routine WRLN changes the buffer to character form and transmits it using a FORTRAN WRITE statement. The reading and writing of individual items then consists of transferring data between stack and buffer. RDCH, for example, gets the next character from the buffer and stores it in the variable whose address is on the top of the stack, while WRCH takes the next-to-top character on the stack and places it in the buffer, using the top stack element as field-width. (This entire I/O system is for Pascal TEXT files only; non-TEXT files require no such buffering and simply use FORTRAN's unformatted I/O statements directly).

RUN-TIME ERROR HANDLER:

Run-time errors in an F-code program are handled by the routine RUNERR, which outputs a message describing the nature and location of the error, and then terminates execution. This error location is given in the form of the (approximate) source program line number at which the error occurred, this form being by far the most helpful. In order that such information be available at run-time, the F-code instructions for every major syntactic construct being translated are preceded by the special instruction:

LINNUM = <current source program line number> .

Note that this also provides a very useful reference for

manually checking the generated F-code against the original Pascal source program — see Appendix B for examples.

MAIN PROGRAM:

The main program for F-code is actually part of the run-time library. As mentioned earlier, it invokes the Pascal main program, which has been translated into the SUBROUTINE R1. It also initializes the stack, etc., as well as preparing files at the start and closing them at the end of program execution.

A study of the run-time library should demonstrate how easily new instructions can be fitted into the F-code framework by simply adding appropriate routines to the library.

Although in general, an intermediate code's instructions tend to be kept semantically weak in order to ease their implementation across a wide range of environments, with F-code there is clearly no such constraint. Even if certain semantic actions which occur together are overall fairly complex, they can nevertheless be identified as "primitive" and implemented by a <u>single</u> F-code instruction (see for example LDKYCH and PREFOR). This helps significantly in keeping F-code programs compact.

Chapter 4 : THE PASCAL-F TRANSLATOR

This chapter first provides a brief overview of the translator, and then gives details of the F-code patterns generated for each Pascal construct.

A. General Description

The Pascal-F translator takes a Pascal program as input and generates an equivalent F-code program as output. It also produces a listing, containing the Pascal source together with diagnostics for any errors which were detected. In order to minimize storage requirements, a second input file is used to hold the (very bulky) text of these error messages, rather than keeping them in an internal array. Lines are then read from this file, as needed, and written onto the listing.

The translator is of course written in Pascal, so that it can ultimately create an F-code version of itself. Although it was initially developed under existing compilers which provided many language extensions, the translator was written with particular attention to ensuring that only standard Pascal was used. Thus no difficulties arose with non-standard features when a self-translation was performed. Furthermore, it was possible to transfer the translator with the greatest ease between three quite different environments (on two continents!) during its development.

With the exception then of a small number of special machine-dependent constants, shown in Fig. 5 below, the entire translator is completely independent of its environment. These constants are gathered together at the very start of the program where they may easily be modified to suit different machines. Essentially, they all depend on only two factors: the machine's wordlength and its character set.

```
const

  MAXINTEGER    =  { Max INTEGER value on this machine          }
  MAXINTDIV10   =  { MAXINTEGER div 10                          }
  MAXINTMOD10   =  { MAXINTEGER mod 10                          }
  MAXORDCHAR    =  { Max ORD value for CHARs                    }
  CHARSPERINT   =  { No of characters per INTEGER storage unit  }
  MAXORDSET     =  { Max ORD value for SET elements             }
  INTSPERSET    =  { (MAXORDSET+1) div (no of bits per INTEGER) }
```

Fig. 5 : Machine-Dependent Constants in the Translator

Because all machine dependencies have been isolated into these constants, generating a copy of the translator for a machine with different characteristics is a very straightforward task. At an existing Pascal-F installation, simply adjust the values of these constants in the Pascal version of the translator and then feed this through the current F-code version. The result will be a new copy of the translator, written in F-code, capable of both running on the target machine and generating suitable F-code for it. (To allow for such "cross-compilation", note that a separate constant MAXINTEGER is used above, rather than the predefined MAXINT).

The overall structure of the Pascal-F translator is basically similar to those of the well-known Pascal-P[12] and Pascal-S[20] translators. The entire translation is carried out in a single pass, the three activities of scanning, parsing, and F-code generation being interleaved. The recursive-descent parsing technique is used, with a separate parsing procedure for each production in Pascal's grammar.

Rather than write the entire Pascal-F translator from scratch, consideration was initially given to modifying Pascal-P to produce F-code rather than P-code. However, this approach was soon rejected for a number of reasons. Substantial changes would be necessary in order to allow for certain aspects of recursion and variable-addressing in F-code; it was felt that the resulting translator then could still be fairly indirect. Moreover, despite its distinguished birthplace, the Pascal-P code itself was found surprisingly difficult to read. Finally, it should be said that the creative prospect of designing and implementing an original program appeared far more attractive than the thought of ploughing through an existing program to isolate and carry out the large number of necessary modifications. It may be of interest to note that the Pascal-F translator eventually contained fewer than half the number of statements occurring in Pascal-P, while also being (hopefully!) quite easy to read and understand.

B. F-code Generation Patterns

For full details of all the F-code instructions referred to below, consult the run-time library listing in Appendix D. Particular instances of most of the following patterns may be found in the sample translations of Appendix C.

Pascal programs consist of declarations and statements. Declarations are essentially directions to the translator to guide the way it analyses and generates code for statements. However, all the declarations in a routine themselves result in the generation of only a single F-code instruction: a call to UPTOP to allocate a block of storage on the stack for all the local variables declared in the routine.

The translation to F-code of both the individual components of statements, as well as the statements themselves, is now considered.

CONSTANTS:

All scalar constants in the Pascal source program, regardless of their data-type, are represented by INTEGERs in F-code using the natural ORD mapping. String constants, on the other hand, are represented by corresponding Hollerith strings. Constants can occur only within expressions; whenever a constant is encountered, a LDVALU (for scalars) or LDSTR (for strings) instruction is generated to load its value onto the stack.

VARIABLES:

As described in the previous chapter, the stack is used to represent all Pascal variables, which are identified by means of their stack addresses. The translator's symbol table stores the declaration level and offset of each variable, which together with the display BASE completely determine its address.

The address of an entire variable is loaded onto the stack by the instruction LDADDR. For a structured variable, an ARRCMP or RECFLD instruction can then be added to load the address of an array component or record field, respectively.

For variables which are formal VAR parameters, an extra dereferencing instruction LDSIMP is used to load the address of the actual parameter, since this address is stored as the formal's value.

The value of a variable is loaded by firstly loading its address and then using LDSIMP (for simple variables) or LDSRUC (for structured variables) to dereference this address.

Field identifiers occurring on their own inside a WITH statement are treated exactly as variables — see the description of WITH below.

FILES:

Although files in Pascal are considered to be variables, it is more convenient to deal with them separately for purposes of code generation. Files are represented in F-code by FORTRAN "logical unit numbers", these being used directly in all I/O

instructions. Logical unit numbers are assigned to Pascal files in the order in which the files are listed in the program heading.

EXPRESSIONS:

Expressions are translated into conventional postfix notation. For each operator, code is first generated to load the value(s) of its operand(s) (this process may in turn involve full expression evaluation), followed by an instruction to apply the operator to these values. Predeclared functions each have a separate F-code instruction, which is generated after the code to load its argument's value. User-declared functions are called in a manner similar to procedure invocation described below. Note that after an expression has been fully evaluated, its result is always left on the top of the stack, all intermediate values having been popped.

ASSIGNMENT STATEMENT:

"VAR := EXPR"

```
load address of VAR
evaluate EXPR
CALL STSIMP / CALL STSRUC(length)  {VAR simple/structured}
```

PROCEDURE CALL STATEMENT:

"P (A1, A2, ..., An)"

```
CALL MKLINK
for each actual parameter Ai in turn,
   load its value/address/routine-number
   as the corresponding formal parameter
   is value/var/routine, respectively
CALL CALLUP(number, level, parsiz)
```

IF STATEMENT:

```
"if EXPR then STMT1 [else STMT2]"

   evaluate EXPR
   IF (FALSE(0)) GOTO L1
   code for STMT1
   GOTO L2            <===  omit these
L1 CONTINUE
   code for STMT2     <===  if there is no
L2 CONTINUE           <===  "else STMT2"
```

CASE STATEMENT:

```
"case EXPR of
   C1        : STMT1;
   C2A, C2B  : STMT2;
   C3        : STMT3
 end"

   evaluate EXPR
   CALL UNLOAD(CASNDX)
   IF (CASNDX .NE. c1) GOTO L2
   code for STMT1
   GOTO L1
L2 IF (CASNDX .NE. c2a) GOTO L3
   GOTO L4
L3 IF (CASNDX .NE. c2b) GOTO L5
L4 CONTINUE
   code for STMT2
   GOTO L1
L5 IF (CASNDX .NE. c3) GOTO L6
   code for STMT3
   GOTO L1
L6 CALL RUNERR(13)  {'CASE index out of range'}
L1 CONTINUE
```

WHILE STATEMENT:

```
"while EXPR do STMT"

L1 CONTINUE
   evaluate EXPR
   IF (FALSE(0)) GOTO L2
   code for STMT
   GOTO L1
L2 CONTINUE
```

REPEAT STATEMENT:

"repeat STMT1; STMT2; ...; STMTn until EXPR"

```
L1 CONTINUE
   code for STMT1
   code for STMT2
   - - - - - - - -
   code for STMTn
   evaluate EXPR
   IF (FALSE(0)) GOTO L1
```

FOR STATEMENT:

"for VAR := EXPR1 to/downto EXPR2 do STMT"

```
   evaluate EXPR1
   evaluate EXPR2
   CALL PREFOR(1/-1, declev, offset, NOLOOP)
   IF (NOLOOP) GOTO L1
L2 CONTINUE
   code for STMT
   CALL TESTEP(MORE, 1/-1)
   IF (MORE) GOTO L2
L1 CONTINUE
```

WITH STATEMENT:

"with REC-VAR do STMT"

```
   load address of REC-VAR
   CALL RECBAS(level)
   code for STMT
   CALL RSTBAS(level)
```

The scope of a WITH statement is essentially a new nesting level in which all the fields of REC-VAR are declared as variables. The base address for this new level is the address of REC-VAR, which is stored in the display by RECBAS; the offset of each variable (i.e. field) is the relative position of the field within REC-VAR. On exit from the WITH statement, the display is restored to its former status by RSTBAS.

Chapter 5 : RESULTS AND CONCLUSIONS

Pascal-F has so far been installed in the following three environments:

- Amdahl 470 running under MTS
- IBM 4341 running under CMS
- VAX-11/780 running under VMS

Although further experiments are needed to test its portability more fully, the results to date have been encouraging and suggest that the Pascal-F system is highly portable. The environment interface is isolated into a small set of constants in the translator and a number of very short and simple routines in the run-time library. Modifying these was quite straightforward in the above cases. By contrast with results for other portable Pascal compilers[10] the transfer of Pascal-F to a new environment involves only a fraction of the effort which is normally required.

Because of time constraints on this project, the current version of the translator does not implement the full Pascal language. A list of the unsupported features is given in Appendix A. With only two exceptions, the implementation of all these missing features has been fully designed, and it was found that they can be grafted on without any fundamental changes to the existing F-code machine and without introducing any further environment-dependencies. A problem arises, however, with internal files, i.e. files not listed in the program heading,

since these do not appear to have any natural implementation within the present framework. Considerable difficulties also occur with implementing a GOTO which jumps out of a routine; in the opinion of many language designers, such a feature should not be allowed in the first place!

Nevertheless, the current language restrictions are not considered to be a serious drawback, since Pascal-F is basically a study in portability, and the translator does possess the crucial ability to perform self-translation.

The translator is a remarkably short program, consisting of some 1,400 Pascal statements only. Its F-code version consists of fewer than 10,000 instructions, indicating a typical Pascal to F-code expansion factor of about 7.

Because the translator is so short and both it and F-code are (hopefully!) easy to read and understand, it is planned to use Pascal-F in an upcoming introductory course on the implementation of programming languages.

An interesting side-effect of the Pascal-F project was the development of a number of useful "extensions" to FORTRAN, particularly the buffered I/O system, which can be used within FORTRAN programs quite independently of Pascal-F. Of course, the entire Pascal-F system may even be viewed as a highly congenial preprocessor for FORTRAN — simply feed in the specifications in the pleasant notation of Pascal, and the

translator will automatically generate the required FORTRAN program!

It has been seen that the overall concept of using an existing high-level language to directly implement an intermediate code has two substantial advantages: portability and extensibility. The chief contribution of the Pascal-F project is towards helping establish the viability of this approach. Because of its widespread availability, FORTRAN was chosen as the target language, despite its serious deficiencies. Even with such constraints, it was still possible to develop a portable translator. With better-suited languages becoming more widely available, the concept of a standard abstract machine based on a high-level language may have an interesting and promising future.

BIBLIOGRAPHY

1 American National Standards Institute [1966] American Standard FORTRAN — X3.9, New York

2 Bourne, S.R., Birrell, A.D., and Walker, I.W. [1975] Z-code, an intermediate object code for Algol 68, The Computer Laboratory, Cambridge.

3 Brandon, D.H. [1977] "Commercial Software", in Software Portability (Editor: Brown, P.J.), Cambridge University Press; chapter VI.D

4 Coleman, S.S., Poole, P.C., and Waite, W.M. [1974] "The Mobile Programming System, Janus" Software — Practice and Experience, Vol 4, pp 5-23

5 Griswold, R.E. [1972] The Macro Implementation of SNOBOL4; A Case Study in Machine-Independent Software Development, W.H. Freeman, San Francisco

6 International Standards Organization [1981] Draft Proposed Pascal Standard, ISO dp 7185

7 Jensen, Kathleen, and Wirth, Niklaus [1974] Pascal User Manual and Report, Springer-Verlag, New York

8 Kernighan, Brian W., and Ritchie, Dennis M. [1978] The C Programming Language, Prentice-Hall, New Jersey

9 Larmouth, J. [1973] "Serious FORTRAN — Part 2" Software — Practice and Experience, Vol 3, pp 197-225

10 Lecarme, Olivier, and Peyrolle-Thomas, Marie-Claude [1978] "Self-compiling Compilers: An Appraisal of their Implementation and Portability" Software — Practice and Experience, Vol 8, pp 149-170

11 Mock, O., Olstzyn, J., Strong, J., Steel, T., Tritter, A., and Wegstein, J. [1958] "The problem of programming communications with changing machines: a proposed solution" Communications of the ACM, Vol 1

12 Nori, K.V., Ammann, U., Jensen, K., and Nägeli, H.H. [1974] The Pascal (P) Compiler: Implementation Notes, Berichte des Instituts für Informatik, No. 10, Eidgenössische Technische Hochschule, Zürich

13 Poole, Peter C. [1976] "Portable and Adaptable Compilers", in Compiler Construction (Editors: Bauer, F.L. and Eickel, J.), Springer-Verlag, New York; chapter 5.A

14 Richards, Martin [1977] "Portable Compilers", in Software Portability (Editor: Brown, P.J.), Cambridge University Press; chapter III.D

15 Richards, Martin, and Whitby-Strevens, Colin [1979] BCPL — the language and its compiler, Cambridge University Press

16 Ridler, Philip F. [1979] A FORTRAN Reference Manual, Pitman, London

17 Steel, T.B. [1961] "A first version of UNCOL" Proc. AFIPS WJCC, Vol 19

18 Weber, L.B. [1973] A Machine Independent Pascal Compiler, M.S. Thesis, Univ. of Colorado, Boulder

19 Wirth, Niklaus [1971] "The Design of a Pascal Compiler" Software — Practice and Experience, Vol 1, pp 309-333

20 Wirth, Niklaus [1975] Pascal-S : A Subset and its Implementation, Berichte des Instituts für Informatik, No. 12, Eidgenössische Technische Hochschule, Zürich

21 Wulf, W., Johnsson, R.K., Weinstock, C.B., Hobbs, S.O., and Geschke, C.M. [1975] The Design of an Optimizing Compiler, American Elsevier, New York

Appendix A : LANGUAGE RESTRICTIONS UNDER PASCAL-F

The following Pascal features are not supported by the current Pascal-F translator:

LABELs
Strings in CONST definitions
The data type REAL
Pointers
Variant records without tag-fields
Multi-dimensional array declarations of the form
  ARRAY [T1, T2, ...]; (use ARRAY [T1] OF ARRAY [T2] OF ...)
Non-TEXT files
Internal files (not listed in the program heading)
The default files INPUT and OUTPUT
File buffers
User-defined FUNCTIONs
The GOTO statement
The operator /
The operators <= and >= for SETs
The form EXPR .. EXPR in SET constructors
Multi-dimensional array subscripts of the form
  [E1, E2, ...]; (use [E1][E2] ...)
Files, PROCEDUREs, or FUNCTIONs as parameters
The predeclared procedures
  DISPOSE GET NEW PACK PAGE PUT UNPACK
The predeclared functions
  ABS ARCTAN COS EXP LN ODD ROUND SIN SQR SQRT TRUNC

In addition, compile-time checks on type-compatibility are incomplete, and run-time range checking (except for CASE index values) is not implemented.

Appendix B : SAMPLE TRANSLATIONS

The F-code generation patterns of the translator are discussed in Chapter 4, section B. These are now illustrated by listing two Pascal programs, each followed by its translation into F-code.

```
===>>  P a s c a l - F   T r a n s l a t o r

 1  program TowersOfHanoi (F);
 2
 3  const
 4     TOTALRINGS = 4;
 5
 6  var
 7     F : TEXT;
 8
 9  {-----------------------------------------------------------}
10  procedure HANOI (RINGS, FROMPEG, TOPEG, WORKPEG : INTEGER);
11
12     begin
13     if RINGS > 0 then
14        begin
15        HANOI(RINGS-1, FROMPEG, WORKPEG, TOPEG);
16        WRITELN(F, FROMPEG:25, TOPEG:9);
17        HANOI(RINGS-1, WORKPEG, TOPEG, FROMPEG)
18        end
19     end {HANOI} ;
20
21  {-----------------------------------------------------------}
22  begin
23  REWRITE(F);
24
25  WRITELN(F, 'Towers of Hanoi for ', TOTALRINGS:1, ' rings');
26  WRITELN(F);
27  WRITELN(F, 'Move top ring from peg _ to peg _');
28
29  HANOI(TOTALRINGS, 1, 2, 3)
30  end .

===>>  No errors detected
```

```
      SUBROUTINE R2
      LOGICAL FALSE,  NOLOOP, MORE
      INTEGER LINNUM, CASNDX, DFLTBW
      COMMON /LINCOM/ LINNUM
                                        LINNUM = 13
      CALL LDADDR(2,3)
      CALL LDSIMP
      CALL LDVALU(0)
      CALL GT
      IF (FALSE(0)) GOTO 1
                                        LINNUM = 15
      CALL MKLINK
      CALL LDADDR(2,3)
      CALL LDSIMP
      CALL LDVALU(1)
      CALL SUB
      CALL LDADDR(2,4)
      CALL LDSIMP
      CALL LDADDR(2,6)
      CALL LDSIMP
      CALL LDADDR(2,5)
      CALL LDSIMP
      CALL CALLUP(2,2,4)
                                        LINNUM = 16
      CALL LDADDR(2,4)
      CALL LDSIMP
      CALL LDVALU(25)
      CALL WRINT(1)
      CALL LDADDR(2,5)
      CALL LDSIMP
      CALL LDVALU(9)
      CALL WRINT(1)
      CALL WRLN(1)
                                        LINNUM = 17
      CALL MKLINK
      CALL LDADDR(2,3)
      CALL LDSIMP
      CALL LDVALU(1)
      CALL SUB
      CALL LDADDR(2,6)
      CALL LDSIMP
      CALL LDADDR(2,5)
      CALL LDSIMP
      CALL LDADDR(2,4)
      CALL LDSIMP
      CALL CALLUP(2,2,4)
    1 CONTINUE
      RETURN
      END
C
C ==============================================================
C
```

```
      SUBROUTINE R1
      LOGICAL FALSE,  NOLOOP, MORE
      INTEGER LINNUM, CASNDX, DFLTBW
      COMMON /LINCOM/ LINNUM
                                  LINNUM = 23
      CALL REWRIT(1)
                                  LINNUM = 25
      CALL LDSTR(5, 20, 20HTowers of Hanoi for )
      CALL LDVALU(20)
      CALL WRSTR(20,1)
      CALL LDVALU(4)
      CALL LDVALU(1)
      CALL WRINT(1)
      CALL LDSTR(2,  6,  8H rings  )
      CALL LDVALU(6)
      CALL WRSTR(6,1)
      CALL WRLN(1)
                                  LINNUM = 26
      CALL WRLN(1)
                                  LINNUM = 27
      CALL LDSTR(9, 33, 36HMove top ring from peg _ to peg _   )
      CALL LDVALU(33)
      CALL WRSTR(33,1)
      CALL WRLN(1)
                                  LINNUM = 29
      CALL MKLINK
      CALL LDVALU(4)
      CALL LDVALU(1)
      CALL LDVALU(2)
      CALL LDVALU(3)
      CALL CALLUP(2,2,4)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE CALLUP (NUMBER, LEVEL, PARSIZ)

      ---  as in Chapter 3, page 28, with n = 2  ---
```

```
===>>  P a s c a l - F   T r a n s l a t o r

 1  program SIEVE (F);
 2
 3  { = = = = = = = = = = = = = = = = = = = = = = = = = = }
 4  {  S i e v e   o f   E r a t o s t h e n e s  }
 5  { = = = = = = = = = = = = = = = = = = = = = = = = = = }
 6
 7  const
 8     TESTLIMIT     = 200;
 9     PRIMESPERLINE = 10;
10
11  var
12     F            : TEXT;
13     ISPRIME      : array [2 .. TESTLIMIT] of BOOLEAN;
14     INDEX,
15     TESTNUMBER,
16     PRIMECOUNT   : INTEGER;
17
18  begin
19  REWRITE(F);
20  WRITELN(F, 'The primes up to ', TESTLIMIT:1, ' are:');
21  PRIMECOUNT := 0;
22
23  { Initialize the sieve }
24  for INDEX := 2 to TESTLIMIT do ISPRIME[INDEX] := TRUE;
25
26  { Calculate and output the primes }
27  for TESTNUMBER := 2 to TESTLIMIT do
28     if ISPRIME[TESTNUMBER] then
29        begin
30
31        { Output and count TESTNUMBER }
32        WRITE(F, TESTNUMBER:6);
33        PRIMECOUNT := PRIMECOUNT + 1;
34        if PRIMECOUNT mod PRIMESPERLINE = 0 then WRITELN(F);
35
36        { Remove all remaining multiples of TESTNUMBER }
37        INDEX := TESTNUMBER * TESTNUMBER;
38        while INDEX <= TESTLIMIT do
39           begin
40           ISPRIME[INDEX] := FALSE;
41           INDEX := INDEX + TESTNUMBER
42           end
43        end;
44
45  WRITELN(F);
46  WRITELN(F, 'The number of primes up to ', TESTLIMIT:1,
47             ' is ', PRIMECOUNT:1)
48  end .

===>>  No errors detected
```

```
      SUBROUTINE R1
      LOGICAL FALSE,  NOLOOP, MORE
      INTEGER LINNUM, CASNDX, DFLTBW
      COMMON /LINCOM/ LINNUM
                                        LINNUM = 18
      CALL UPTOP(202)
                                        LINNUM = 19
      CALL REWRIT(1)
                                        LINNUM = 20
      CALL LDSTR(5, 17, 20HThe primes up to     )
      CALL LDVALU(18)
      CALL WRSTR(18,1)
      CALL LDVALU(200)
      CALL LDVALU(1)
      CALL WRINT(1)
      CALL LDSTR(2,  5,  8H are:    )
      CALL LDVALU(5)
      CALL WRSTR(5,1)
      CALL WRLN(1)
                                        LINNUM = 21
      CALL LDADDR(1,204)
      CALL LDVALU(0)
      CALL STSIMP
                                        LINNUM = 24
      CALL LDVALU(2)
      CALL LDVALU(200)
      CALL PREFOR(1,1,202,NOLOOP)
      IF (NOLOOP) GOTO 1
    2 CONTINUE
                                        LINNUM = 24
      CALL LDADDR(1,3)
      CALL LDADDR(1,202)
      CALL LDSIMP
      CALL ARRCMP(2,1)
      CALL LDVALU(1)
      CALL STSIMP
      CALL TESTEP(MORE,1)
      IF (MORE) GOTO 2
    1 CONTINUE
                                        LINNUM = 27
      CALL LDVALU(2)
      CALL LDVALU(200)
      CALL PREFOR(1,1,203,NOLOOP)
      IF (NOLOOP) GOTO 3
    4 CONTINUE
                                        LINNUM = 28
      CALL LDADDR(1,3)
      CALL LDADDR(1,203)
      CALL LDSIMP
      CALL ARRCMP(2,1)
      CALL LDSIMP
      IF (FALSE(0)) GOTO 5
```

```
                                        LINNUM = 32
      CALL LDADDR(1,203)
      CALL LDSIMP
      CALL LDVALU(6)
      CALL WRINT(1)
                                        LINNUM = 33
      CALL LDADDR(1,204)
      CALL LDADDR(1,204)
      CALL LDSIMP
      CALL LDVALU(1)
      CALL ADD
      CALL STSIMP
                                        LINNUM = 34
      CALL LDADDR(1,204)
      CALL LDSIMP
      CALL LDVALU(10)
      CALL MOD
      CALL LDVALU(0)
      CALL EQ
      IF (FALSE(0)) GOTO 6
                                        LINNUM = 34
      CALL WRLN(1)
    6 CONTINUE
                                        LINNUM = 37
      CALL LDADDR(1,202)
      CALL LDADDR(1,203)
      CALL LDSIMP
      CALL LDADDR(1,203)
      CALL LDSIMP
      CALL MUL
      CALL STSIMP
                                        LINNUM = 38
    7 CONTINUE
      CALL LDADDR(1,202)
      CALL LDSIMP
      CALL LDVALU(200)
      CALL LE
      IF (FALSE(0)) GOTO 8
                                        LINNUM = 40
      CALL LDADDR(1,3)
      CALL LDADDR(1,202)
      CALL LDSIMP
      CALL ARRCMP(2,1)
      CALL LDVALU(0)
      CALL STSIMP
                                        LINNUM = 41
      CALL LDADDR(1,202)
      CALL LDADDR(1,202)
      CALL LDSIMP
      CALL LDADDR(1,203)
      CALL LDSIMP
      CALL ADD
```

```
      CALL STSIMP
      GOTO 7
    8 CONTINUE
    5 CONTINUE
      CALL TESTEP(MORE,1)
      IF (MORE) GOTO 4
    3 CONTINUE
                                    LINNUM = 45
      CALL WRLN(1)
                                    LINNUM = 46
      CALL LDSTR(7, 27, 28HThe number of primes up to  )
      CALL LDVALU(27)
      CALL WRSTR(27,1)
      CALL LDVALU(200)
      CALL LDVALU(1)
      CALL WRINT(1)
      CALL LDSTR(1,  4,  4H is )
      CALL LDVALU(4)
      CALL WRSTR(4,1)
      CALL LDADDR(1,204)
      CALL LDSIMP
      CALL LDVALU(1)
      CALL WRINT(1)
      CALL WRLN(1)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE CALLUP (NUMBER, LEVEL, PARSIZ)

      ---  as in Chapter 3, page 28, with n = 1  ---
```

Appendix C : LISTING OF THE RUN-TIME LIBRARY

A complete listing of the run-time library is given on the following pages. A general overview of its organization and purpose may be found in Chapter 3, section E. The environment interface routines are those for the Amdahl 470 V8 running under the Michigan Terminal System (MTS) with the FORTRAN H (21.8/4.1) compiler.

The listing below has been re-formatted somewhat from the actual program, in order that it may fit onto these pages.

```
C ************************************************************
C *   P a s c a l - F    R u n t i m e    L i b r a r y    *
C *   ==================================================    *
C *                                                         *
C *                                                         *
C *   List  of  Routines :                                  *
C *   --------------------                                  *
C *                                                         *
C *       BLOCK DATA   (Machine  Interface)                 *
C *       UPKCWD       (Machine  Interface)                 *
C *       PKSWD        (Machine  Interface)                 *
C *       UPKSWD       (Machine  Interface)                 *
C *                                                         *
C *       OPENRD       (Operating  System  Interface)       *
C *       OPENWR       (Operating  System  Interface)       *
C *       GETLN        (Operating  System  Interface)       *
C *       WRNULN       (Operating  System  Interface)       *
C *                                                         *
C *       UNPKSR                                            *
C *                                                         *
C *       UPTOP                                             *
C *       MKLINK                                            *
C *                                                         *
C *       LDVALU                                            *
C *       LDADDR                                            *
C *       LDSIMP                                            *
C *       LDSRUC                                            *
C *       LDSTR                                             *
C *       LDKYCH                                            *
C *                                                         *
C *       STSIMP                                            *
C *       STSRUC                                            *
C *       COPY                                              *
C *                                                         *
C *       RECBAS                                            *
C *       RSTBAS                                            *
C *       ARRCMP                                            *
C *       RECFLD                                            *
C *                                                         *
C *       ADD                                               *
C *       SUB                                               *
C *       MUL                                               *
C *       DIV                                               *
C *       MOD                                               *
C *                                                         *
C *       NOT                                               *
C *       AND                                               *
C *       OR                                                *
C *                                                         *
C *       UNION                                             *
C *       INTER                                             *
C *       DIFF                                              *
```

```
C *        IN                                                    *
C *        MKSET                                                 *
C *        PKSET                                                 *
C *        UPKSET                                                *
C *                                                              *
C *        EQ                                                    *
C *        NE                                                    *
C *        LT                                                    *
C *        LE                                                    *
C *        GT                                                    *
C *        GE                                                    *
C *        MKRELN                                                *
C *                                                              *
C *        PRED                                                  *
C *        SUCC                                                  *
C *        EOF                                                   *
C *        EOLN                                                  *
C *                                                              *
C *        FALSE                                                 *
C *        UNLOAD                                                *
C *                                                              *
C *        PREFOR                                                *
C *        TESTEP                                                *
C *                                                              *
C *        SETFLS                                                *
C *        CLSFLS                                                *
C *        CLSFIL                                                *
C *                                                              *
C *        RESET                                                 *
C *        GET                                                   *
C *        RDCH                                                  *
C *        RDINT                                                 *
C *        DGTEST                                                *
C *        RDLN                                                  *
C *                                                              *
C *        REWRIT                                                *
C *        PUT                                                   *
C *        WRCH                                                  *
C *        WRINT                                                 *
C *        WRBOOL                                                *
C *        DFLTBW                                                *
C *        WRSTR                                                 *
C *        WRLN                                                  *
C *                                                              *
C *        RUNERR                                                *
C *                                                              *
C *        Main Program                                          *
C *                                                              *
C *                                                              *
C *    Machine - Dependent  Constants :                          *
C *    --------------------------------                          *
C *                                                              *
```

```
C *       MAXORDCHAR + 1  (size of char set)                          *
C *          = no of elts in CHR                                      *
C *       CHR(n) = Pascal's CHR(n-1), 1 <= n <= MAXORDCHAR+1          *
C *                                                                   *
C *       ORDSPC = ORD(' ')                                           *
C *       ORDPLS = ORD('+')                                           *
C *       ORDMNS = ORD('-')                                           *
C *       ORD0   = ORD('0')                                           *
C *       ORD9   = ORD('9')                                           *
C *       MXDV10 = MAXINT div 10                                      *
C *       MXMD10 = MAXINT mod 10                                      *
C *       SAVESZ = size of save area                                  *
C *                                                                   *
C *       CHARSPERINT  (no of chars per INTEGER storage unit)         *
C *          = no of elts in second param of routine UPKCWD           *
C *          = no of elts in UWORD in routine UNPKSR                  *
C *          = no used to set UWDPOS in routine UNPKSR                *
C *          = used to set lengths in calls to LDSTR in WRBOOL        *
C *                                                                   *
C *       WORDLENGTH  (no of bits per INTEGER storage unit)           *
C *       MAXORDSET   (max ORD value for SET elements)                *
C *       INTSPERSET  ((MAXORDSET+1) div WORDLENGTH)                  *
C *          = constants used in                                      *
C *            UNION, INTER, DIFF, IN, MKSET, PKSET, UPKSET           *
C *                                                                   *
C *       MAXLINELEN  (max line length for TEXT files)                *
C *          = no of rows in LINE                                     *
C *          = no of elts in fourth param of GETLN                    *
C *          = overflow limit on LINLEN in GET and PUT                *
C *          = no of A1 fields in FORMAT stmt in WRLN                 *
C *                                                                   *
C *   Translator - Dependent  Constants :                             *
C *   ----------------------------------                              *
C *                                                                   *
C *       MAXFILES  (max no of files permitted)                       *
C *          = no of cols in LINE                                     *
C *          = no of elts in FILSTS, LINLEN, RDPOSN,                  *
C *                          WINDOW, ENDFIL, ENDLIN                   *
C *          = upper limit of FILNUM in SETFLS and CLSFLS             *
C *                                                                   *
C *       MAXSTACKSIZE  (max size of STACK)                           *
C *          = no of elts in STACK                                    *
C *          = overflow limit in routine UPTOP                        *
C *                                                                   *
C *       MAXLEVEL  (max static nesting of routines)                  *
C *          = no of elts in BASE                                     *
C *          = upper limit of BASNUM in Main Program                  *
C *                                                                   *
C *       F = 0 = ORD(FALSE)                                          *
C *       T = 1 = ORD(TRUE)                                           *
C *                                                                   *
C *********************************************************************
```

```
C
   BLOCK DATA
C
C  Machine Interface Routine
C  Defines the values of the FILCOM constants:
C     R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
C  Version here for:
C     All machines using the EBCDIC char set
C
   INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
  + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
  + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
   COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
  + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
   DATA R/0/, W/1/, U/-1/,  F/0/, T/1/
C
   DATA
  + CHR(  1)/1H /, CHR(  2)/1H /, CHR(  3)/1H /, CHR(  4)/1H /,
  + CHR(  5)/1H /, CHR(  6)/1H /, CHR(  7)/1H /, CHR(  8)/1H /,
  + CHR(  9)/1H /, CHR( 10)/1H /, CHR( 11)/1H /, CHR( 12)/1H /,
  + CHR( 13)/1H /, CHR( 14)/1H /, CHR( 15)/1H /, CHR( 16)/1H /,
  + CHR( 17)/1Hδ/, CHR( 18)/1Hε/, CHR( 19)/1Hζ/, CHR( 20)/1Hη/,
  + CHR( 21)/1Hθ/, CHR( 22)/1Hι/, CHR( 23)/1Hκ/, CHR( 24)/1Hλ/,
  + CHR( 25)/1Hμ/, CHR( 26)/1Hν/, CHR( 27)/1Hξ/, CHR( 28)/1Hο/,
  + CHR( 29)/1Hπ/, CHR( 30)/1Hρ/, CHR( 31)/1Hσ/, CHR( 32)/1Hτ/,
  + CHR( 33)/1Hυ/, CHR( 34)/1Hφ/, CHR( 35)/1Hχ/, CHR( 36)/1Hψ/,
  + CHR( 37)/1Hω/, CHR( 38)/1HΓ/, CHR( 39)/1HΔ/, CHR( 40)/1HΘ/,
  + CHR( 41)/1HΛ/, CHR( 42)/1HΞ/, CHR( 43)/1HΠ/, CHR( 44)/1HΣ/,
  + CHR( 45)/1HΤ/, CHR( 46)/1HΦ/, CHR( 47)/1HΨ/, CHR( 48)/1HΩ/,
  + CHR( 49)/1H /, CHR( 50)/1H /, CHR( 51)/1H /, CHR( 52)/1H /,
  + CHR( 53)/1H /, CHR( 54)/1H /, CHR( 55)/1H /, CHR( 56)/1H /,
  + CHR( 57)/1H /, CHR( 58)/1H /, CHR( 59)/1H /, CHR( 60)/1H /,
  + CHR( 61)/1H /, CHR( 62)/1H /, CHR( 63)/1H /, CHR( 64)/1H /,
  + CHR( 65)/1H /, CHR( 66)/1Hþ/, CHR( 67)/1HŁ/, CHR( 68)/1HØ/,
  + CHR( 69)/1HÐ/, CHR( 70)/1HÞ/, CHR( 71)/1HÆ/, CHR( 72)/1HŒ/,
  + CHR( 73)/1H®/, CHR( 74)/1H·/, CHR( 75)/1H¢/, CHR( 76)/1H./
   DATA
  + CHR( 77)/1H<⁄, CHR( 78)/1H(/, CHR( 79)/1H+/, CHR( 80)/1H|/,
  + CHR( 81)/1H&/, CHR( 82)/1H₁/, CHR( 83)/1H₂/, CHR( 84)/1H₃/,
  + CHR( 85)/1H₄/, CHR( 86)/1H₅/, CHR( 87)/1H₆/, CHR( 88)/1H₇/,
  + CHR( 89)/1H₈/, CHR( 90)/1H₉/, CHR( 91)/1H!/, CHR( 92)/1H$/,
  + CHR( 93)/1H*/, CHR( 94)/1H)/, CHR( 95)/1H;/, CHR( 96)/1H¬/,
  + CHR( 97)/1H-/, CHR( 98)/1H//, CHR( 99)/1Hł/, CHR(100)/1Hø/,
  + CHR(101)/1Hđ/, CHR(102)/1Hþ/, CHR(103)/1Hæ/, CHR(104)/1Hœ/,
  + CHR(105)/1Hı/, CHR(106)/1H"/, CHR(107)/1H¦/, CHR(108)/1H,/,
  + CHR(109)/1H%/, CHR(110)/1H_/, CHR(111)/1H>/, CHR(112)/1H?/,
  + CHR(113)/1H°/, CHR(114)/1H¯/, CHR(115)/1H¨/, CHR(116)/1H~/,
  + CHR(117)/1H´/, CHR(118)/1H˙/, CHR(119)/1H¸/, CHR(120)/1H˘/,
  + CHR(121)/1H˘/, CHR(122)/1H`/, CHR(123)/1H:/, CHR(124)/1H#/,
  + CHR(125)/1H@/, CHR(126)/1H'/, CHR(127)/1H=/, CHR(128)/1H"/,
```

```
     + CHR(129)/1H₀/, CHR(130)/1Ha/, CHR(131)/1Hb/, CHR(132)/1Hc/,
     + CHR(133)/1Hd/, CHR(134)/1He/, CHR(135)/1Hf/, CHR(136)/1Hg/,
     + CHR(137)/1Hh/, CHR(138)/1Hi/, CHR(139)/1H⁷/, CHR(140)/1H{/,
     + CHR(141)/1H≤/, CHR(142)/1H⁽/, CHR(143)/1H⁺/, CHR(144)/1H┼/,
     + CHR(145)/1H /, CHR(146)/1Hj/, CHR(147)/1Hk/, CHR(148)/1Hl/,
     + CHR(149)/1Hm/, CHR(150)/1Hn/, CHR(151)/1Ho/, CHR(152)/1Hp/
      DATA
     + CHR(153)/1Hq/, CHR(154)/1Hr/, CHR(155)/1H‡/, CHR(156)/1H}/,
     + CHR(157)/1H¤/, CHR(158)/1H⁾/, CHR(159)/1H±/, CHR(160)/1H■/,
     + CHR(161)/1H⁻/, CHR(162)/1H°/, CHR(163)/1Hs/, CHR(164)/1Ht/,
     + CHR(165)/1Hu/, CHR(166)/1Hv/, CHR(167)/1Hw/, CHR(168)/1Hx/,
     + CHR(169)/1Hy/, CHR(170)/1Hz/, CHR(171)/1Hα/, CHR(172)/1H└/,
     + CHR(173)/1H┌/, CHR(174)/1H[/, CHR(175)/1H≥/, CHR(176)/1H•/,
     + CHR(177)/1H⁰/, CHR(178)/1H¹/, CHR(179)/1H²/, CHR(180)/1H³/,
     + CHR(181)/1H⁴/, CHR(182)/1H⁵/, CHR(183)/1H⁶/, CHR(184)/1H⁷/,
     + CHR(185)/1H⁸/, CHR(186)/1H⁹/, CHR(187)/1H⁻/, CHR(188)/1H┘/,
     + CHR(189)/1H┐/, CHR(190)/1H]/, CHR(191)/1H≠/, CHR(192)/1H—/,
     + CHR(193)/1H /, CHR(194)/1HA/, CHR(195)/1HB/, CHR(196)/1HC/,
     + CHR(197)/1HD/, CHR(198)/1HE/, CHR(199)/1HF/, CHR(200)/1HG/,
     + CHR(201)/1HH/, CHR(202)/1HI/, CHR(203)/1H₊/, CHR(204)/1H.̇/,
     + CHR(205)/1H₎/, CHR(206)/1H₍/, CHR(207)/1H'/, CHR(208)/1H‘/,
     + CHR(209)/1H /, CHR(210)/1HJ/, CHR(211)/1HK/, CHR(212)/1HL/,
     + CHR(213)/1HM/, CHR(214)/1HN/, CHR(215)/1HO/, CHR(216)/1HP/,
     + CHR(217)/1HQ/, CHR(218)/1HR/, CHR(219)/1H©/, CHR(220)/1H£/,
     + CHR(221)/1Hβ/, CHR(222)/1Hγ/, CHR(223)/1H /, CHR(224)/1H /,
     + CHR(225)/1H\/, CHR(226)/1H /, CHR(227)/1HS/, CHR(228)/1HT/
      DATA
     + CHR(229)/1HU/, CHR(230)/1HV/, CHR(231)/1HW/, CHR(232)/1HX/,
     + CHR(233)/1HY/, CHR(234)/1HZ/, CHR(235)/1H̥/, CHR(236)/1H_/,
     + CHR(237)/1H†/, CHR(238)/1H`/, CHR(239)/1H'/, CHR(240)/1H¯/,
     + CHR(241)/1H0/, CHR(242)/1H1/, CHR(243)/1H2/, CHR(244)/1H3/,
     + CHR(245)/1H4/, CHR(246)/1H5/, CHR(247)/1H6/, CHR(248)/1H7/,
     + CHR(249)/1H8/, CHR(250)/1H9/, CHR(251)/1H^/, CHR(252)/1H~/,
     + CHR(253)/1H /, CHR(254)/1H /, CHR(255)/1H /, CHR(256)/1H /
C
      DATA
     + ORDSPC/64/, ORDPLS/78/, ORDMNS/96/, ORD0/240/, ORD9/249/,
     + MXDV10/214748364/, MXMD10/7/
      END
C
C ==============================================================
C
      SUBROUTINE UPKCWD (PWORD, UWORD)
C
C  Machine Interface Routine
C  Unpack a single word of chars
C  Input Parameter:
C     PWORD - an INTEGER variable, containing CHARSPERINT
C             chars packed in Hollerith format
C  Output Parameter:
C     UWORD - an array of CHARSPERINT INTEGERs, containing
C             the ORD values of these chars
```

```
C
C  Version here for:
C     CHARSPERINT=4,  32-bit word,  2's complement arithmetic,
C     chars packed in word from high-order to low-order end
C        e.g.  IBM 4341,  IBM 370,  Amdahl 470
C
   INTEGER PWORD, UWORD(4), MAXINT, CHRCNT, UWDPOS
   LOGICAL NEGATV
   DATA MAXINT/2147483647/
C
   NEGATV = (PWORD .LT. 0)
   IF (NEGATV) PWORD = (PWORD + MAXINT) + 1
   DO 1 CHRCNT = 1, 4
      UWDPOS = 5 - CHRCNT
      UWORD(UWDPOS) = MOD(PWORD, 256)
 1    PWORD = PWORD / 256
   IF (NEGATV) UWORD(1) = UWORD(1) + 128
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE PKSWD (UWORD, PWORD)
C
C  Machine Interface Routine
C  Pack individual bits into a single word
C  Input Parameter:
C     UWORD - an array of WORDLENGTH INTEGERs,
C             containing individual bit values
C  Output Parameter:
C     PWORD - a single INTEGER variable,
C             containing the packed bits of UWORD
C
C  Version here for:
C     32-bit word,  2's complement arithmetic
C        e.g.  IBM 4341,  IBM 370,  Amdahl 470
C
   INTEGER UWORD(32), PWORD, MAXINT, UPOS
   DATA MAXINT/2147483647/
C
   PWORD = 0
   DO 1 UPOS = 2, 32
 1    PWORD = 2 * PWORD + UWORD(UPOS)
   IF (UWORD(1) .EQ. 1) PWORD = (PWORD - MAXINT) - 1
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE UPKSWD (PWORD, UWORD)
C
C  Machine Interface Routine
```

```
C  Unpack a word into individual bits
C  Input Parameter:
C     PWORD - an INTEGER variable
C  Output Parameter:
C     UWORD - an array of WORDLENGTH INTEGERs,
C             containing the individual bits of PWORD
C
C  Version here for:
C     32-bit word, 2's complement arithmetic
C        e.g.  IBM 4341,  IBM 370,  Amdahl 470
C
   INTEGER PWORD, UWORD(32), MAXINT, BITCNT, UPOS
   DATA MAXINT/2147483647/
C
   IF (PWORD .GE. 0) GOTO 1
      UWORD(1) = 1
      PWORD = (PWORD + MAXINT) + 1
      GOTO 2
 1 CONTINUE
      UWORD(1) = 0
 2 DO 3 BITCNT = 2, 32
      UPOS = 34 - BITCNT
      UWORD(UPOS) = MOD(PWORD, 2)
 3    PWORD = PWORD / 2
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE OPENRD (FILNUM)
C
C  Operating System Interface Routine
C  Open a file for reading, by positioning to its start
C  Input Parameter:
C     FILNUM - the INTEGER logical unit number of the file
C
C  Version here for:
C     MTS on Amdahl 470 V8 at UBC, with FORTRAN H compiler
C
   INTEGER FILNUM
C
   REWIND FILNUM
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE OPENWR (FILNUM)
C
C  Operating System Interface Routine
C  Open a file for writing, by emptying it
C  Input Parameter:
```

```
C     FILNUM - the INTEGER logical unit number of the file
C
C  Version here for:
C     MTS on Amdahl 470 V8 at UBC, with FORTRAN H compiler
C     EMPTYS is an MTS System Subroutine that empties a file
C
   INTEGER FILNUM
C
   CALL EMPTYS(FILNUM)
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE GETLN (FILNUM, ENDFIL, LNLEN, LN)
C
C  Operating System Interface Routine
C  Attempt to read a line of chars
C  Input Parameter:
C     FILNUM - the INTEGER logical unit no of the input file
C  Output Parameters:
C     ENDFIL - an INTEGER variable;  if a line was obtained,
C              set to 0 (= F),  otherwise set to 1 (= T)
C     LNLEN  - an INTEGER variable;  if a line was obtained,
C              set to the number of chars on the line
C     LN     - an array of MAXLINELEN INTEGERs;  if a line
C              was obtained and had at most MAXLINELEN chars,
C              stores the ORD values of these chars
C
C  Version here for:
C     MTS on Amdahl 470 V8 at UBC, with FORTRAN H compiler
C     READ is an MTS System Subroutine that reads a single line
C
   INTEGER FILNUM, ENDFIL, LNLEN, LN(254)
   INTEGER PKDLN(64), RDMOD, F, T, LINNUM, INTLEN
   INTEGER*2 CHRLEN(3)
   DATA RDMOD/Z08004000/, F/0/, T/1/
C
   CHRLEN(2) = 256
   CALL READ(PKDLN, CHRLEN, RDMOD, LINNUM, FILNUM, &1)
      ENDFIL = F
      LNLEN  = CHRLEN(1)
      IF (LNLEN .GT. 254) GOTO 2
         INTLEN = (LNLEN + 3) / 4
         CALL UNPKSR(PKDLN, INTLEN, LNLEN, LN)
         GOTO 2
 1 CONTINUE
      ENDFIL = T
 2 RETURN
   END
C
C ==============================================================
```

```
C
   SUBROUTINE WRNULN (FILNUM)
C
C  Operating System Interface Routine
C  Output a null (empty) line
C  Input Parameter:
C     FILNUM - the INTEGER logical unit no of the output file
C
C  Version here for:
C     MTS on Amdahl 470 V8 at UBC, with FORTRAN H compiler
C
   INTEGER FILNUM
C
   WRITE(FILNUM,1)
   RETURN
 1 FORMAT( )
   END
C
C ==============================================================
C
   SUBROUTINE UNPKSR (PSTR, INTLEN, CHRLEN, USTR)
C
C  Unpack a string of chars
C  Input Parameters:
C     PSTR   - an array of INTEGERs, contains a Hollerith string
C     INTLEN - the no of elts (INTEGERs) in PSTR
C     CHRLEN - the no of chars packed in PSTR (left-justified)
C  Output Parameter:
C     USTR   - an array of CHRLEN INTEGERs, contains
C              the ORD values of these chars
C
   INTEGER INTLEN, CHRLEN, PSTR(INTLEN), USTR(CHRLEN)
   INTEGER PPOS, UPOS, PWORD, UWORD(4), UWDPOS
C
   PPOS = 0
   DO 1 UPOS = 1, CHRLEN
      UWDPOS = MOD(UPOS-1, 4) + 1
      IF (UWDPOS .NE. 1) GOTO 1
         PPOS  = PPOS + 1
         PWORD = PSTR(PPOS)
         CALL UPKCWD(PWORD, UWORD)
 1    USTR(UPOS) = UWORD(UWDPOS)
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE UPTOP (INCREM)
C
C  Increment STACK pointer TOP by INCREM, checking for overflow
C
   INTEGER INCREM
```

```
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      TOP = TOP + INCREM
      IF (TOP .GT. 5000) CALL RUNERR(1)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE MKLINK
C
C  Make space on top of STACK for routine linkage info
C
      INTEGER STACK(5000), TOP, BASE(30), SAVESZ, LINKSZ
      COMMON  STACK, TOP, BASE, SAVESZ, LINKSZ
C
      CALL UPTOP(LINKSZ)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE LDVALU (VALU)
C
C  Load onto STACK the value VALU
C
      INTEGER VALU
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      CALL UPTOP(1)
      STACK(TOP) = VALU
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE LDADDR (DECLEV, OFFSET)
C
C  Load onto STACK the address  BASE(DECLEV) + OFFSET
C
      INTEGER DECLEV, OFFSET
      INTEGER STACK(5000), TOP, BASE(30)
      COMMON  STACK, TOP, BASE
C
      CALL UPTOP(1)
      STACK(TOP) = BASE(DECLEV) + OFFSET
      RETURN
      END
C
C ==============================================================
```

```
C
   SUBROUTINE LDSIMP
C
C  Load onto STACK the simple value at
C  address STACK(TOP), popping this address
C
   INTEGER ADDR
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   ADDR = STACK(TOP)
   STACK(TOP) = STACK(ADDR)
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE LDSRUC (LENGTH)
C
C  Load onto STACK the structured value of length LENGTH,
C  which starts at address STACK(TOP), popping this address
C
   INTEGER LENGTH, SRCADR, DSTADR
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   SRCADR = STACK(TOP)
   DSTADR = TOP
   CALL UPTOP(LENGTH-1)
   CALL COPY(LENGTH, SRCADR, DSTADR)
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE LDSTR (INTLEN, CHRLEN, PSTR)
C
C  Load onto STACK an unpacked copy of the string PSTR
C     PSTR is an array of INTLEN INTEGERs, containing a
C     left-justified Hollerith string of CHRLEN chars
C
   INTEGER INTLEN, CHRLEN, PSTR(INTLEN), STRADR
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   STRADR = TOP + 1
   CALL UPTOP(CHRLEN)
   CALL UNPKSR(PSTR, INTLEN, CHRLEN, STACK(STRADR))
   RETURN
   END
C
C ==============================================================
```

```
C
      SUBROUTINE LDKYCH (LENGTH)
C
C  Load onto STACK the key characters for comparison of strings
C  On entry, there are two char strings each of length LENGTH
C     on top of STACK
C  On exit, the strings have been popped; on top of STACK are
C     a pair of chars (ORD values) from the respective strings;
C     either the first pair of unequal chars from corresponding
C     string positions, or the final char in each string
C
      INTEGER LENGTH, BEGPOS, POS, SR1POS, SR2POS
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      BEGPOS = TOP - 2*LENGTH + 1
      POS    = 1
      SR1POS = BEGPOS
      SR2POS = BEGPOS + LENGTH
    1 IF (POS.EQ.LENGTH .OR. STACK(SR1POS).NE.STACK(SR2POS)) GOTO 2
         POS    = POS + 1
         SR1POS = SR1POS + 1
         SR2POS = SR2POS + 1
         GOTO 1
C
    2 STACK(BEGPOS)   = STACK(SR1POS)
      STACK(BEGPOS+1) = STACK(SR2POS)
      TOP = BEGPOS + 1
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE STSIMP
C
C  Store the simple value which is on the top of STACK
C  in the location whose address is stored just below it,
C  and  then  pop  both  the  value  and  the  address
C
      INTEGER ADDR
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      ADDR = STACK(TOP-1)
      STACK(ADDR) = STACK(TOP)
      TOP = TOP - 2
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE STSRUC (LENGTH)
```

```
C
C  Store the structured value of length LENGTH which is on the
C  top of STACK in the region whose starting address is stored
C  just below it,  and then pop both the value and the address
C
   INTEGER LENGTH, ADRPOS
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   ADRPOS = TOP - LENGTH
   CALL COPY(LENGTH, ADRPOS+1, STACK(ADRPOS))
   TOP = ADRPOS - 1
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE COPY (LENGTH, SRCADR, DSTADR)
C
C  Copy LENGTH entries  from address SRCADR  to address DSTADR
C
   INTEGER LENGTH, SRCADR, DSTADR, POS, SRCPOS, DSTPOS
   INTEGER STACK(5000)
   COMMON  STACK
C
   DO 1 POS = 1, LENGTH
      SRCPOS = SRCADR - 1 + POS
      DSTPOS = DSTADR - 1 + POS
 1    STACK(DSTPOS) = STACK(SRCPOS)
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE RECBAS (LEVEL)
C
C  Swap the contents of BASE(LEVEL) and STACK(TOP)
C
   INTEGER LEVEL, OLDBSL
   INTEGER STACK(5000), TOP, BASE(30)
   COMMON  STACK, TOP, BASE
C
   OLDBSL       = BASE(LEVEL)
   BASE(LEVEL)  = STACK(TOP)
   STACK(TOP)   = OLDBSL
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE RSTBAS (LEVEL)
C
```

```
C  Restore BASE(LEVEL) from STACK(TOP); pop STACK
C
   INTEGER LEVEL
   INTEGER STACK(5000), TOP, BASE(30)
   COMMON  STACK, TOP, BASE
C
   BASE(LEVEL) = STACK(TOP)
   TOP = TOP - 1
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE ARRCMP (MINORD, COMPSZ)
C
C  Compute the address of an array component, from
C     STACK(TOP-1) : address of start of array
C     STACK(TOP)   : value of index expression
C     MINORD       : index of first array component
C     COMPSZ       : size of each array component
C  Pop STACK, and store the result in (new) STACK(TOP)
C
   INTEGER MINORD, COMPSZ
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   STACK(TOP-1) = STACK(TOP-1) + (STACK(TOP) - MINORD) * COMPSZ
   TOP = TOP - 1
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE RECFLD (FLDOFS)
C
C  Compute the address of a record field, from
C     STACK(TOP) : address of start of record
C     FLDOFS     : field offset within record
C  Store the result in STACK(TOP)
C
   INTEGER FLDOFS
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   STACK(TOP) = STACK(TOP) + FLDOFS
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE ADD
C
```

```
C  Store (STACK(TOP-1) + STACK(TOP)) in STACK(TOP-1); pop STACK
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   STACK(TOP-1) = STACK(TOP-1) + STACK(TOP)
   TOP = TOP - 1
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE SUB
C
C  Store (STACK(TOP-1) - STACK(TOP)) in STACK(TOP-1); pop STACK
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   STACK(TOP-1) = STACK(TOP-1) - STACK(TOP)
   TOP = TOP - 1
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE MUL
C
C  Store (STACK(TOP-1) * STACK(TOP)) in STACK(TOP-1); pop STACK
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   STACK(TOP-1) = STACK(TOP-1) * STACK(TOP)
   TOP = TOP - 1
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE DIV
C
C  Store (STACK(TOP-1) div STACK(TOP))in STACK(TOP-1); pop STACK
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   IF (STACK(TOP) .EQ. 0) CALL RUNERR(10)
   STACK(TOP-1) = STACK(TOP-1) / STACK(TOP)
   TOP = TOP - 1
   RETURN
   END
```

```
C
C ==============================================================
C
      SUBROUTINE MOD
C
C     Store (STACK(TOP-1) mod STACK(TOP))in STACK(TOP-1); pop STACK
C
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      IF (STACK(TOP) .LE. 0) CALL RUNERR(11)
      STACK(TOP-1) = MOD(STACK(TOP-1), STACK(TOP))
      IF (STACK(TOP-1).LT.0) STACK(TOP-1) = STACK(TOP-1)+STACK(TOP)
      TOP = TOP - 1
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE NOT
C
C     Store (not STACK(TOP)) in STACK(TOP)
C
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      STACK(TOP) = 1 - STACK(TOP)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE AND
C
C     Store (STACK(TOP-1) and STACK(TOP))in STACK(TOP-1); pop STACK
C
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      STACK(TOP-1) = STACK(TOP-1) * STACK(TOP)
      TOP = TOP - 1
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE OR
C
C     Store (STACK(TOP-1) or STACK(TOP)) in STACK(TOP-1); pop STACK
C
      INTEGER STACK(5000), TOP
```

```
      COMMON  STACK, TOP
C
      STACK(TOP-1) = MIN0(STACK(TOP-1) + STACK(TOP), 1)
      TOP = TOP - 1
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE UNION
C
C  Replace the top two sets on STACK by their union
C
      INTEGER USET1(256), USET2(256), RESULT(256), ELTPOS
C
      CALL UPKSET(USET2)
      CALL UPKSET(USET1)
      DO 1 ELTPOS = 1, 256
 1       RESULT(ELTPOS) = MIN0(USET1(ELTPOS) + USET2(ELTPOS), 1)
      CALL PKSET(RESULT)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE INTER
C
C  Replace the top two sets on STACK by their intersection
C
      INTEGER USET1(256), USET2(256), RESULT(256), ELTPOS
C
      CALL UPKSET(USET2)
      CALL UPKSET(USET1)
      DO 1 ELTPOS = 1, 256
 1       RESULT(ELTPOS) = USET1(ELTPOS) * USET2(ELTPOS)
      CALL PKSET(RESULT)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE DIFF
C
C  Replace the top two sets on STACK by their difference
C    (next-to-top set) - (top set)
C
      INTEGER USET1(256), USET2(256), RESULT(256), ELTPOS
C
      CALL UPKSET(USET2)
      CALL UPKSET(USET1)
      DO 1 ELTPOS = 1, 256
 1       RESULT(ELTPOS) = MAX0(USET1(ELTPOS) - USET2(ELTPOS), 0)
```

```
      CALL PKSET(RESULT)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE IN
C
C  Determine if the set on top of STACK contains the
C  element stored just below it; pop set and element
C
      INTEGER ELT, RESULT, WDPOS, BITPOS, UWORD(32)
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      ELT = STACK(TOP-8)
      IF (ELT .GE. 0  .AND.  ELT .LE. 255) GOTO 1
         RESULT = 0
         GOTO 2
    1 CONTINUE
         WDPOS  = TOP - 8 + (ELT / 32) + 1
         CALL UPKSWD(STACK(WDPOS), UWORD)
         BITPOS = MOD(ELT, 32) + 1
         RESULT = UWORD(BITPOS)
    2 TOP = TOP - 8
      STACK(TOP) = RESULT
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE MKSET (ELTCNT)
C
C  Replace the top ELTCNT STACK entries,
C  each 0 or 1, by the corresponding set
C
      INTEGER ELTCNT, USET(256), UPOS, BEGPOS, STKPOS, ELT
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      DO 1 UPOS = 1, 256
    1    USET(UPOS) = 0
      IF (ELTCNT .EQ. 0) GOTO 3
         BEGPOS = TOP - ELTCNT + 1
         DO 2 STKPOS = BEGPOS, TOP
            ELT = STACK(STKPOS)
    2       USET(ELT+1) = 1
         TOP = TOP - ELTCNT
C
    3 CALL PKSET(USET)
      RETURN
      END
```

```
C
C ==============================================================
C
      SUBROUTINE PKSET (USET)
C
C  Pack the elements of USET, each 0 or 1,
C  into a set, and load this onto STACK
C
      INTEGER USET(256), WORD, STKPOS
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      CALL UPTOP(8)
      DO 1 WORD = 1, 8
         STKPOS = TOP - 8 + WORD
 1       CALL PKSWD(USET(32*WORD-31), STACK(STKPOS))
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE UPKSET (USET)
C
C  Unpack the set on top of STACK into USET, popping the set
C
      INTEGER USET(256), WORD, STKPOS
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      DO 1 WORD = 1, 8
         STKPOS = TOP - 8 + WORD
 1       CALL UPKSWD(STACK(STKPOS), USET(32*WORD-31))
      TOP = TOP - 8
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE EQ
C
C  Store (STACK(TOP-1) = STACK(TOP)) in STACK(TOP-1); pop STACK
C
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      CALL MKRELN(STACK(TOP-1) .EQ. STACK(TOP))
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE NE
```

```
C
C  Store (STACK(TOP-1) <> STACK(TOP)) in STACK(TOP-1); pop STACK
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   CALL MKRELN(STACK(TOP-1) .NE. STACK(TOP))
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE LT
C
C  Store (STACK(TOP-1) < STACK(TOP)) in STACK(TOP-1); pop STACK
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   CALL MKRELN(STACK(TOP-1) .LT. STACK(TOP))
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE LE
C
C  Store (STACK(TOP-1) <= STACK(TOP)) in STACK(TOP-1); pop STACK
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   CALL MKRELN(STACK(TOP-1) .LE. STACK(TOP))
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE GT
C
C  Store (STACK(TOP-1) > STACK(TOP)) in STACK(TOP-1); pop STACK
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   CALL MKRELN(STACK(TOP-1) .GT. STACK(TOP))
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE GE
```

```
C
C  Store (STACK(TOP-1) >= STACK(TOP)) in STACK(TOP-1); pop STACK
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   CALL MKRELN(STACK(TOP-1) .GE. STACK(TOP))
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE MKRELN (RELN)
C
C  Pop STACK, and store ORD(RELN) in (new) STACK(TOP)
C
   LOGICAL RELN
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   TOP = TOP - 1
   IF (RELN) GOTO 1
      STACK(TOP) = 0
      GOTO 2
 1 CONTINUE
      STACK(TOP) = 1
 2 RETURN
   END
C
C ==============================================================
C
   SUBROUTINE PRED
C
C  Store STACK(TOP)-1 in STACK(TOP)
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   STACK(TOP) = STACK(TOP) - 1
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE SUCC
C
C  Store STACK(TOP)+1 in STACK(TOP)
C
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   STACK(TOP) = STACK(TOP) + 1
```

```
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE EOF (FILNUM)
C
C  Load onto STACK the value of EOF(FILNUM)
C
      INTEGER FILNUM
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      IF (FILSTS(FILNUM) .EQ. U) CALL RUNERR(12)
      CALL LDVALU(ENDFIL(FILNUM))
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE EOLN (FILNUM)
C
C  Load onto STACK the value of EOLN(FILNUM)
C
      INTEGER FILNUM
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      IF (FILSTS(FILNUM) .EQ. U) CALL RUNERR(12)
      IF (ENDFIL(FILNUM) .EQ. T) CALL RUNERR(12)
      CALL LDVALU(ENDLIN(FILNUM))
      RETURN
      END
C
C ==============================================================
C
      LOGICAL FUNCTION FALSE (NOPARM)
C
C  Return  STACK(TOP) .EQ. 0; pop STACK
C
      INTEGER NOPARM
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      FALSE = (STACK(TOP) .EQ. 0)
      TOP   = TOP - 1
```

```
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE UNLOAD (VALUE)
C
C  Store the value of STACK(TOP) in VALUE; pop STACK
C
      INTEGER VALUE
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      VALUE = STACK(TOP)
      TOP   = TOP - 1
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE PREFOR (STEP, DECLEV, OFFSET, NOLOOP)
C
C  Prepare  to  execute  a  FOR  loop
C  On entry, STACK(TOP-1) and STACK(TOP) contain the FOR loop's
C  evaluated "initial-value" and "final-value", respectively.
C  If the range between these two values is empty, then
C     set NOLOOP to .TRUE.
C     pop these top two values
C  else
C     set NOLOOP to .FALSE.
C     set the control variable to "initial-value",
C        leaving its address in STACK(TOP)
C
      INTEGER STEP, DECLEV, OFFSET, INLVAL, FNLVAL, ADDR
      LOGICAL NOLOOP
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      INLVAL = STACK(TOP-1)
      FNLVAL = STACK(TOP)
      NOLOOP = (STEP .EQ. +1  .AND.  INLVAL .GT. FNLVAL) .OR.
     +         (STEP .EQ. -1  .AND.  INLVAL .LT. FNLVAL)
C
      IF (NOLOOP) GOTO 1
         CALL LDADDR(DECLEV, OFFSET)
         ADDR = STACK(TOP)
         STACK(ADDR) = INLVAL
         GOTO 2
    1 CONTINUE
         TOP = TOP - 2
    2 RETURN
      END
```

```
C
C ==============================================================
C
      SUBROUTINE TESTEP (MORE, STEP)
C
C  Test / Step  the  control  variable  of  a  FOR  loop
C  On entry, STACK(TOP-2), STACK(TOP-1) and STACK(TOP)
C  contain the FOR loop's "initial-value", "final-value",
C  and  control  variable's  address,  respectively.
C  If the control variable's value equals "final-value", then
C     set MORE to .FALSE.
C     pop the address, "final-value" and "initial-value"
C  else
C     set MORE to .TRUE.
C     alter the control variable's value by STEP
C
      LOGICAL MORE
      INTEGER STEP, ADDR
      INTEGER STACK(5000), TOP
      COMMON  STACK, TOP
C
      ADDR = STACK(TOP)
      MORE = (STACK(ADDR) .NE. STACK(TOP-1))
C
      IF (MORE) GOTO 1
         TOP = TOP - 3
         GOTO 2
    1 CONTINUE
         STACK(ADDR) = STACK(ADDR) + STEP
    2 RETURN
      END
C
C ==============================================================
C
      SUBROUTINE SETFLS
C
C  Initialize  FILSTS to U  and  WINDOW to ORDSPC  for each file
C
      INTEGER FILNUM
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      DO 1 FILNUM = 1, 6
         FILSTS(FILNUM) = U
    1    WINDOW(FILNUM) = ORDSPC
      RETURN
      END
C
C ==============================================================
```

```
C
   SUBROUTINE CLSFLS
C
C  Issue a closing WRITELN, where needed, for each output file
C
   INTEGER FILNUM
   INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
  + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
  + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
   COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
  + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
   DO 1 FILNUM = 1, 6
 1    CALL CLSFIL(FILNUM)
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE CLSFIL (FILNUM)
C
C  Issue a closing WRITELN, if needed, for output file FILNUM
C
   INTEGER FILNUM
   INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
  + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
  + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
   COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
  + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
   IF (FILSTS(FILNUM) .NE. W) GOTO 1
      IF (LINLEN(FILNUM) .GT. 0) CALL WRLN(FILNUM)
 1 RETURN
   END
C
C ==============================================================
C
   SUBROUTINE RESET (FILNUM)
C
C  Implements the Pascal procedure RESET
C  Prepare file FILNUM for reading  by positioning to its start
C  and placing its first component to the buffer WINDOW(FILNUM)
C
   INTEGER FILNUM
   INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
  + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
  + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
   COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
  + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
   CALL CLSFIL(FILNUM)
   CALL OPENRD(FILNUM)
```

```
      FILSTS(FILNUM) = R
      ENDFIL(FILNUM) = F
      ENDLIN(FILNUM) = T
      CALL GET(FILNUM)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE GET (FILNUM)
C
C  Implements the Pascal procedure GET
C  Advance to next component of file FILNUM; place it in buffer
C  WINDOW(FILNUM); also set ENDFIL(FILNUM) and ENDLIN(FILNUM)
C
      INTEGER FILNUM, INDEX
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      IF (FILSTS(FILNUM) .NE. R) CALL RUNERR(2)
      IF (ENDFIL(FILNUM) .EQ. T) CALL RUNERR(3)
C
      IF (ENDLIN(FILNUM) .EQ. F) GOTO 1
         CALL GETLN(FILNUM, ENDFIL(FILNUM),
     +              LINLEN(FILNUM), LINE(1,FILNUM))
         IF (ENDFIL(FILNUM) .EQ. T) GOTO 3
            IF (LINLEN(FILNUM) .GT. 254) CALL RUNERR(4)
            RDPOSN(FILNUM) = 0
    1 IF (RDPOSN(FILNUM) .LT. LINLEN(FILNUM)) GOTO 2
         ENDLIN(FILNUM) = T
         WINDOW(FILNUM) = ORDSPC
         GOTO 3
    2 CONTINUE
         ENDLIN(FILNUM) = F
         RDPOSN(FILNUM) = RDPOSN(FILNUM) + 1
         INDEX = RDPOSN(FILNUM)
         WINDOW(FILNUM) = LINE(INDEX, FILNUM)
    3 RETURN
      END
C
C ==============================================================
C
      SUBROUTINE RDCH (FILNUM)
C
C  Read from file FILNUM a char, and store
C  its ORD value in STACK(STACK(TOP)); pop STACK
C
      INTEGER FILNUM, ADDR
      INTEGER STACK(5000), TOP
```

```
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON STACK, TOP
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      ADDR = STACK(TOP)
      TOP  = TOP - 1
      STACK(ADDR) = WINDOW(FILNUM)
      CALL GET(FILNUM)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE RDINT (FILNUM)
C
C  Read from file FILNUM an integer, and store
C  its value in STACK(STACK(TOP)); pop STACK
C
      INTEGER FILNUM, INT, DGTVAL, ADDR
      LOGICAL NEGATV, ISDGT
      INTEGER STACK(5000), TOP
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON STACK, TOP
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      IF (FILSTS(FILNUM) .NE. R) CALL RUNERR(2)
C
    1 IF (WINDOW(FILNUM) .NE. ORDSPC) GOTO 2
         CALL GET(FILNUM)
         GOTO 1
    2 NEGATV = (WINDOW(FILNUM) .EQ. ORDMNS)
      IF (NEGATV .OR. WINDOW(FILNUM).EQ.ORDPLS) CALL GET(FILNUM)
      CALL DGTEST(WINDOW(FILNUM), ISDGT, DGTVAL)
      IF (.NOT. ISDGT) CALL RUNERR(5)
C
      INT = 0
    3 CONTINUE
         IF (INT.GT.MXDV10 .OR.
     +       INT.EQ.MXDV10 .AND. DGTVAL.GT.MXMD10) CALL RUNERR(6)
         INT = 10 * INT + DGTVAL
         CALL GET(FILNUM)
         CALL DGTEST(WINDOW(FILNUM), ISDGT, DGTVAL)
         IF (ISDGT) GOTO 3
      IF (NEGATV) INT = -INT
C
      ADDR = STACK(TOP)
```

```
      TOP  = TOP - 1
      STACK(ADDR) = INT
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE DGTEST (TSTORD, ISDGT, DGTVAL)
C
C  Test if a given char represents a decimal digit,
C  and if so, return the numeric value of the digit
C
      INTEGER TSTORD, DGTVAL
      LOGICAL ISDGT
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      ISDGT = (TSTORD.GE.ORD0 .AND. TSTORD.LE.ORD9)
      IF (ISDGT) DGTVAL = TSTORD - ORD0
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE RDLN (FILNUM)
C
C  Implements the (single-parameter) Pascal procedure READLN
C  Skip to the start of the next line, or EOF, on file FILNUM
C
      INTEGER FILNUM
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      ENDLIN(FILNUM) = T
      CALL GET(FILNUM)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE REWRIT (FILNUM)
C
C  Implements the Pascal procedure REWRITE
C  Discard current value of file FILNUM; prepare it for writing
C
      INTEGER FILNUM
```

```
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      CALL CLSFIL(FILNUM)
      CALL OPENWR(FILNUM)
      FILSTS(FILNUM) = W
      ENDFIL(FILNUM) = T
      LINLEN(FILNUM) = 0
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE PUT (FILNUM)
C
C  Implements the Pascal procedure PUT
C  Append the buffer WINDOW(FILNUM) to file FILNUM
C
      INTEGER FILNUM, INDEX
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      IF (FILSTS(FILNUM) .NE. W)   CALL RUNERR(7)
      IF (LINLEN(FILNUM) .EQ. 254) CALL RUNERR(8)
C
      LINLEN(FILNUM) = LINLEN(FILNUM) + 1
      INDEX = LINLEN(FILNUM)
      LINE(INDEX, FILNUM) = WINDOW(FILNUM)
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE WRCH (FILNUM)
C
C  Write onto file FILNUM the char (ORD value) in STACK(TOP-1),
C  using the field-width in STACK(TOP); pop the top two entries
C
      INTEGER FILNUM, ORDCH, WIDTH, SPACES, SPCCNT
      INTEGER STACK(5000), TOP
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON STACK, TOP
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
```

```
C
   WIDTH = STACK(TOP)
   ORDCH = STACK(TOP-1)
   TOP   = TOP - 2
C
   IF (WIDTH .LE. 0) CALL RUNERR(9)
   IF (WIDTH .EQ. 1) GOTO 2
      SPACES = WIDTH - 1
      DO 1 SPCCNT = 1, SPACES
         WINDOW(FILNUM) = ORDSPC
 1       CALL PUT(FILNUM)
C
 2 WINDOW(FILNUM) = ORDCH
   CALL PUT(FILNUM)
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE WRINT (FILNUM)
C
C  Write onto file FILNUM the integer in STACK(TOP-1), using
C  the field-width in STACK(TOP); pop the top two entries
C
   INTEGER FILNUM, INT, WIDTH, ABSINT, ABSLEN, INTLEN,
  +                            SPACES, SPCCNT, DGTCNT
   INTEGER STACK(5000), TOP
   INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
  + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
  + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
   COMMON STACK, TOP
   COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
  + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
   WIDTH = STACK(TOP)
   INT   = STACK(TOP-1)
   TOP   = TOP - 2
C
   IF (WIDTH .LE. 0) CALL RUNERR(9)
   ABSINT = IABS(INT)
   ABSLEN = 1
 1 IF (ABSINT .LT. 10) GOTO 2
      ABSLEN = ABSLEN + 1
      ABSINT = ABSINT / 10
      GOTO 1
C
 2 IF (INT .GE. 0) INTLEN = ABSLEN
   IF (INT .LT. 0) INTLEN = ABSLEN + 1
   IF (WIDTH .LE. INTLEN) GOTO 4
      SPACES = WIDTH - INTLEN
      DO 3 SPCCNT = 1, SPACES
         WINDOW(FILNUM) = ORDSPC
```

```
 3        CALL PUT(FILNUM)
C
 4 IF (INT .GE. 0) GOTO 5
      WINDOW(FILNUM) = ORDMNS
      CALL PUT(FILNUM)
 5 ABSINT = IABS(INT)
   DO 6 DGTCNT = 1, ABSLEN
      WINDOW(FILNUM) = ORD0+MOD(ABSINT/(10**(ABSLEN-DGTCNT)),10)
 6    CALL PUT(FILNUM)
   RETURN
   END
C
C ==============================================================
C
   SUBROUTINE WRBOOL (FILNUM)
C
C  Write onto file FILNUM the Boolean in STACK(TOP-1), using
C  the field-width in STACK(TOP); pop the top two entries
C
   INTEGER FILNUM, BOOL, WIDTH, STRLEN
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   WIDTH = STACK(TOP)
   BOOL  = STACK(TOP-1)
   TOP   = TOP - 2
C
   IF (BOOL .EQ. 1) GOTO 1
      CALL LDSTR(2, 5, 8HFALSE    )
      STRLEN = 5
      GOTO 2
 1 CONTINUE
      CALL LDSTR(1, 4, 4HTRUE)
      STRLEN = 4
 2 CALL LDVALU(WIDTH)
   CALL WRSTR(STRLEN, FILNUM)
   RETURN
   END
C
C ==============================================================
C
   INTEGER FUNCTION DFLTBW (NOPARM)
C
C  Returns the default field-width for the Boolean in STACK(TOP)
C
   INTEGER NOPARM
   INTEGER STACK(5000), TOP
   COMMON  STACK, TOP
C
   DFLTBW = 5 - STACK(TOP)
   RETURN
   END
```

```
C
C ==============================================================
C
      SUBROUTINE WRSTR (STRLEN, FILNUM)
C
C  Write onto file FILNUM the char string of length STRLEN,
C  stored in STACK finishing at position TOP-1, using the
C  field-width in STACK(TOP); pop the string and field-width
C
      INTEGER STRLEN,FILNUM,WIDTH,SPACES,SPCCNT,LOPOS,HIPOS,STKPOS
      INTEGER STACK(5000), TOP
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON STACK, TOP
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
      WIDTH = STACK(TOP)
      LOPOS = TOP - STRLEN
C
      IF (WIDTH .LE. 0) CALL RUNERR(9)
C
      IF (WIDTH .LE. STRLEN) GOTO 2
         SPACES = WIDTH - STRLEN
         DO 1 SPCCNT = 1, SPACES
            WINDOW(FILNUM) = ORDSPC
 1          CALL PUT(FILNUM)
C
 2    HIPOS = LOPOS - 1 + MIN0(STRLEN, WIDTH)
      DO 3 STKPOS = LOPOS, HIPOS
         WINDOW(FILNUM) = STACK(STKPOS)
 3       CALL PUT(FILNUM)
C
      TOP = TOP - 1 - STRLEN
      RETURN
      END
C
C ==============================================================
C
      SUBROUTINE WRLN (FILNUM)
C
C  Implements the (single-parameter) Pascal procedure WRITELN
C  Output a line onto file FILNUM  and empty the line buffer
C
      INTEGER FILNUM, WRLEN, WRPOSN, WRORD
      INTEGER LINE(254,6), FILSTS(6), LINLEN(6), RDPOSN(6),
     + WINDOW(6), ENDFIL(6), ENDLIN(6), R, W, U, F, T,
     + CHR(256), ORDSPC, ORDPLS, ORDMNS, ORD0, ORD9, MXDV10, MXMD10
      COMMON/FILCOM/LINE,FILSTS,LINLEN,RDPOSN,WINDOW,ENDFIL,ENDLIN,
     + R,W,U,F,T,CHR,ORDSPC,ORDPLS,ORDMNS,ORD0,ORD9,MXDV10,MXMD10
C
```

```
      IF (FILSTS(FILNUM) .NE. W) CALL RUNERR(7)
C
      WRLEN = LINLEN(FILNUM)
      IF (WRLEN .GT. 0) GOTO 1
         CALL WRNULN(FILNUM)
         GOTO 3
    1 CONTINUE
         DO 2 WRPOSN = 1, WRLEN
            WRORD = LINE(WRPOSN, FILNUM)
    2       LINE(WRPOSN, FILNUM) = CHR(WRORD + 1)
         WRITE(FILNUM,4) (LINE(WRPOSN, FILNUM), WRPOSN = 1, WRLEN)
         LINLEN(FILNUM) = 0
    3 RETURN
    4 FORMAT(254A1)
      END
C
C ==============================================================
C
      SUBROUTINE RUNERR (ERRNUM)
C
C  Issue run-time error ERRNUM  and halt execution
C
         INTEGER ERRNUM
         INTEGER LINNUM
         COMMON /LINCOM/ LINNUM
C
         CALL CLSFLS
C
         WRITE(6,100) LINNUM
  100    FORMAT(/ 27H *****   Run Error at line , I5)
         GOTO (1,2,3,4,5,6,7,8,9,10,11,12,13,998), ERRNUM
C
    1    WRITE(6,101)
  101    FORMAT(23H *****   Stack overflow /)
         GOTO 999
    2    WRITE(6,102)
  102    FORMAT(40H *****   GET/READ attempted before RESET /)
         GOTO 999
    3    WRITE(6,103)
  103    FORMAT(41H *****   Attempt to read past end-of-file /)
         GOTO 999
    4    WRITE(6,104)
  104    FORMAT(28H *****   Input line too long /)
         GOTO 999
    5    WRITE(6,105)
  105    FORMAT(32H *****   Digit expected on input /)
         GOTO 999
    6    WRITE(6,106)
  106    FORMAT(32H *****   Input integer too large /)
         GOTO 999
    7    WRITE(6,107)
  107    FORMAT(43H *****   PUT/WRITE attempted before REWRITE /)
```

```
      GOTO 999
    8 WRITE(6,108)
  108 FORMAT(29H *****   Output line too long /)
      GOTO 999
    9 WRITE(6,109)
  109 FORMAT(40H *****   Output field-width non-positive /)
      GOTO 999
   10 WRITE(6,110)
  110 FORMAT(38H *****   Second operand of DIV is zero /)
      GOTO 999
   11 WRITE(6,111)
  111 FORMAT(41H *****   Second operand of MOD is <= zero /)
      GOTO 999
   12 WRITE(6,112)
  112 FORMAT(30H *****   EOF or EOLN undefined /)
      GOTO 999
   13 WRITE(6,113)
  113 FORMAT(32H *****   CASE index out of range /)
      GOTO 999
C
  999 STOP
  998 RETURN
      END
C
C ==============================================================
C
C  PROGRAM MAIN
C
C  The  Main  Program
C
   INTEGER BASNUM
   INTEGER STACK(5000), TOP, BASE(30), SAVESZ, LINKSZ, BASEXC
   COMMON STACK, TOP, BASE, SAVESZ, LINKSZ, BASEXC
C
   SAVESZ = 19
   LINKSZ = SAVESZ + 3
   DO 1 BASNUM = 1,20
 1 BASE(BASNUM) = -1
   BASEXC = -1
   TOP    = 0
   CALL SETFLS
C
   CALL MKLINK
   CALL CALLUP(1,1,0)
C
   CALL CLSFLS
   STOP
   END
```