Pascal and C through a Translator Program

Pascal and C through a Translator Program

Zensho NAKAO* and Peter ARATHOON*

Abstract

An automatic translation program was developed to simulate a Pascal compiler which translates Pascal source programs to C target programs; the program runs under the UNIX operating system. Through the process of translation, several distinct differences were found between the two languages which had often been compared elsewhere from different points of view. The differences form the difficulties which necessarily arise and which must be resolved to achieve a faithful translation effect-wise between programs in these high-level programming languages.

PASCAL —> C:WHY?

Initially this research work was a project in using the 'C' language. Translators from one language to another are needed whenever there is a compiler available for a computer for one language but not for another. Instead of acquiring the appropriate compiler, it is possible to make a translator program and use the existing compiler. In the case of the Panafacom computer at the University of the Ryukyus, the Unicus system does not have a Pascal compiler, and so the translation project would have a definite practical purpose. 'C' is particularly suited to writing compiler program systems, and so this project is also realistic in its use of 'C'. It will be seen that Pascal and 'C' are very similar languages, but also that they have sufficient differences to make the translation far from trivial.

REVIEW OF PASCAL

The structure of a Pascal program is basically one large program with any number of nested programs inside it. Each subprogram may have similar subprograms, and is the same as the main program in structure, apart from the leading lines of each, which define whether they are procedures, functions or the main program itself.

This largest basic structure is the Block, written for the main program and for any procedures or functions within it. At the head of each block is a section defining LABELs, (for use with " GOTO " statements as line-markers), CONSTants, (their identifiers and fixed values valid for the rest of that block, this scope being the same for variables), and VARiables, singly or mutiply defined as to their type. One more set of definitions is headed

Received: October 31, 1985

*Department of Electronics and Information Engineering, Faculty of Engineering

by TYPE, this being used mostly for string definitions.

Once the following procedures and functions have been declared, the main group of execution statements occurs, embraced by a BEGIN and END, and separated by semicolons.

Statements in Pascal are similar to those in 'C' and expressions are similar in Pascal to most languages. Symbols and words are both included as operators, (e.g. DIV and OR). The syntax groups Type, Simple Type and Field List need not be included explicitly as separate compiler-functions, but rather may be included in other such functions. Expression testing must of course ve included, to check source expressions before translation.

Two very important statements essential for input and output, READ and WRITE, and their associated READLN and WRITELN, are available functions on Pascal. These too must be handled by the program, which must write them in an economical way in 'C'.

PASCAL'S PROCEDURES AND FUNCTIONS

Pascal has two sorts of subprogram, procedures and functions. Procedures may be sent variables to manipulate, but may not return any value to the calling function, Functions on the other hand MAY return values, The type of the value they return has to be declared for all functions in the function declaration's first line. Variables may be sent by value or by address. If sent by address, the type of the variable is introduced by the word "var" in the procedure/function's parameter list, e. g.:

function F (a, b:real; var c: character) :real;

where a, b are sent by value to F, but the character is sent by address, and the value it has in the calling function will be changed permanently by any change affected on c in this function F. N.B. the way the function's return type is declared after the parameter list. A similar procedure declaration would look like:

procedure P (a,b:real; var c: character) ;

since it may not return any value.

Thus in Pascal the action of a subprogram may be clearer (but sometimes more limited) than in 'C' because (1)subprograms may only be called in the program or subprogram in which they are defined; (2)if it is returning any value it will be a function not a procedure.

REVIEW OF 'C'

The major difference perhaps between 'C' and Pascal is that 'C' has its main program and all subprograms are written separately, (even in different files). where Pascal's are all written together in nested form.

'C's functions may be called from any other functions, not from just a limited set of others; (hiding function names IS possible, but not a general procedure). 'C's functions are not divided into those returning values and those not; they either have an expression in a RETURN statement, or they have an empty or missing RETURN statement. If they return float values, (equivalent to Pascal's real type), then they have to declare so in their declaration and in any function calling them. Otherwise all functions are assumed to return either integers or characters, (whose types are equivalent in 'C' for this purpose). Strings are easily sent to functions, and changes made to them are permanent, since sending a string, (just by its name, with no other symbols necessary), is actually only sending the address of its first element. For sending other types to functions by address, the addresses have to be explicitly defined in the functionas pointers to appropiate memory space:

```
function call:          fn (&a,&b) ;      (&:"address of.. ")
function definition:    fn (px,py)
                        int  *px, *py;
                        {
                        .....
                        }
```

Constants in 'C' may be initialized as variables, assigned their constant values and used just like that, as long as their values are not changed. They are not protected against change if written like this, (e. g. definition of variable b in function F shown below). A better way is to #define constants, e.g.

```
#define P I          3.14
```

This definition must be placed above a function declaration line, whether the function is "main ( )" or a subprogram. Wherever P I is found, from its declaration through to the end of the file it is written in, it will be replaced by the value 3.14, and is protected against inadvertent change, because to assign a new value to it requires an address, i. e., pointers must be used.

Variables are declared similarly to Pascal:

PASCAL	'C'
a, b:integer;	int a,b;
c:character;	char c;
d:real;	float d;

But these definitions may come:

(A) external to the function, just below any #define statements,

(B) first thing in the function, or

(C) first thing in any compound statement inside a function.

For example:

```
# define  P I     3. 14
int a;

F (m, n)
int m, n;
{
```

```
float b=10.0;
    ...

while (b>0){
    char c;
    ....
    ....
    }

....
....
....
}
```

(A) allows variables to exist from that point to the end of the file, (i. e., it can be called from any other following functions, inluding those compiled separately). (B) and (C) bring variables into existence only as far as the closing '} ' of the compound statement they are defined in.

Pascal's constants may be translated as # define statements; variables may become external variables, (type (A) above), and subprogram parameter list variables may become automatic variables, (type (B) above).

DECLARATION OF VARIABLES IN PASCAL

In Pascal, the scope of variables is the subprogram in which they are defined. This means that they are global until the subprogram defining them is left. This poses a problem for their translation into 'C'. In 'C' functions may not be defined inside other functions. The scope of variables is, as in Pascal, limited to their own block, but in Pascal this may cover several fns/procs nested within the subprogram. The easiest solution is to make all variables external in 'C'. It might be possible to send higher-declared variables "down" to lower functions using them, but this would not be worthwhile. With this translation then, there will be problems because of the extra legal scope of variables in the translation than they had in the original. Perhaps the biggest user restriction is that no two variables may have the same name, (because both are external in 'C'). Accidental occurrence in the source program of variables declared in a previous but LOWER fn/proc, which would not be acceptable in Pascal, would be accepted by the 'C' compiler if translated without checking whether that variable should be available at that point in the program. So a check is necessary to see if a variable is allowed or not. This may be done by keeping a memory of all declared variables whose scope extends to the point of translation in the program. This must be reinitialized on leaving every block, variables declared in that block being deleted.

DIFFERENCES BETWEEN 'C' AND PASCAL

(1) Basic program structure

Where PASCAL has a nested system of subprograms, 'C' has each one written separately.

(2) Variable types and scope of variables

Pascal has constants and variables, whereas 'C' has # define statements and automatic/external variables. The former are far wider in application than just substituting values for constant expressions, and may be used as mini-functions in themselves. 'C' also has other types, but these are not considered in this treatment of the translation of simpler PASCAL capabilities.

(3) Procedures, functions and their scope

In Pascal there are procedures and functions, the latter returning values to the calling function. In 'C' there are only functions, which cover the capabilities of procedures and functions, but are not so restricted as to what they return. One function may, for example, be considered to return either an integer or a character, depending on the way the value is treated in the calling function, (e.g., sometimes returning a character, sometimes a #defined constant such as FALSE. In fact both of these are returned in the same form).

In Pascal subprograms may only be called from the block in which they are defined, whereas in 'C' there is no such limitation unless you specifically make it. Thus the compiler/translator must check each call to subprograms for their validity—("May this subprogram be called from this position?")—and only then write it in 'C'.

(4) WITH_DO and IN

Most statements have similar ones in 'C' but these are peculiar to Pascal.

(5) IF_ELSE

Compare Pascal's and 'C's IF_ELSE statements:

```
Pascal:    1)  if (exp) then (statement) else (statememnt)
  'C':     1)  if (exp) (statement);
           2)  else (statement);
```

It can be seen that Pascal's single statement, (which need not have the ELSE), is written in 'C' as two separate statements which need braces { } around them to be treated by the 'C'-compiler as a compound statement.

This means that whenever IF...ELSE... is found in the Pascal source program the translator must write "{if...else... }". Rather than do this, the braces are written in the translation around ALL statements, even if single ones.

(6) Compound statements

In Pascal, BEGIN...END are used to group several statements together into a compound one, as mentioned in (5) above. In 'C' this function is done by braces { ....}. If these are automatically written wherever a statement, single or compound, may be expected, the compiler only has to check the presence of the delimiters, not to actually translate them.

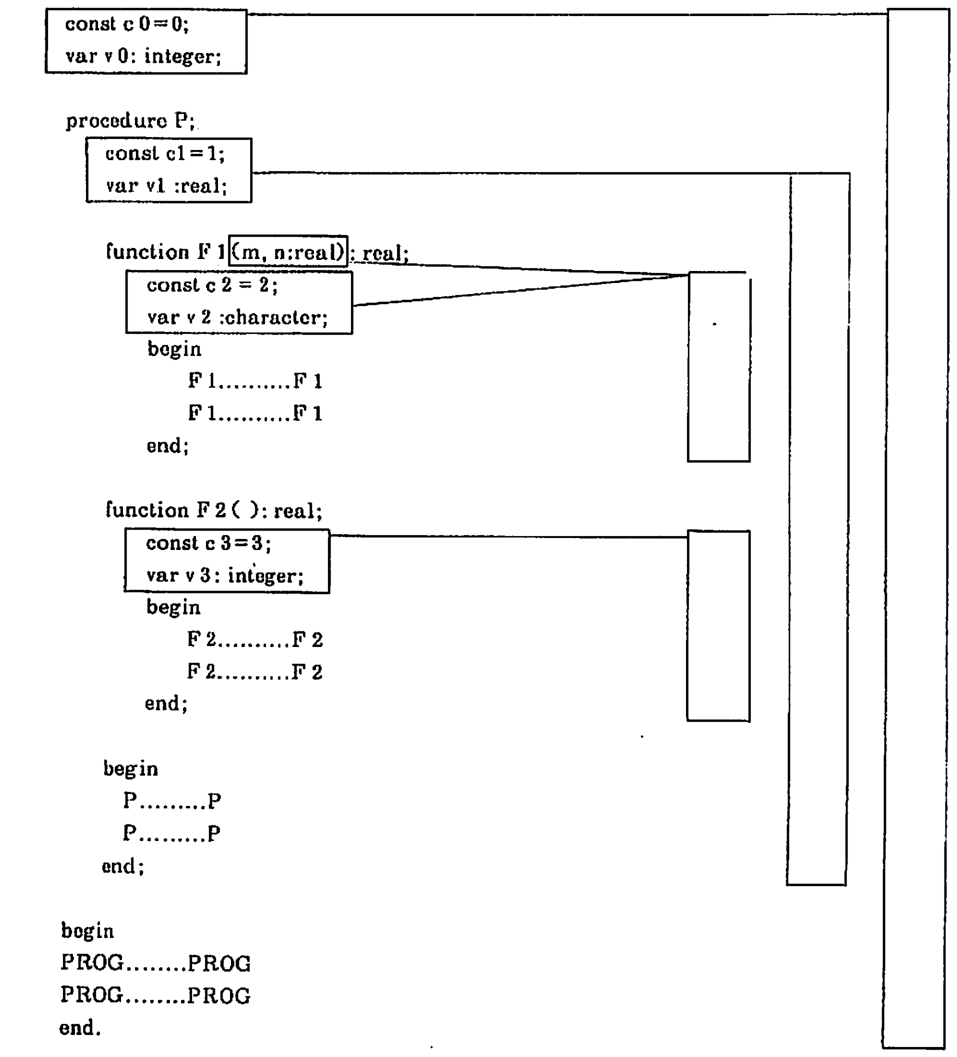
(7) Input/output statements

Pascal's READ and WRITE calls to these predefined functions contain only quoted

EXAMPLES COMPARING THE SCOPE OF VARIABLES IN 'C' AND PASCAL

A) Pascal

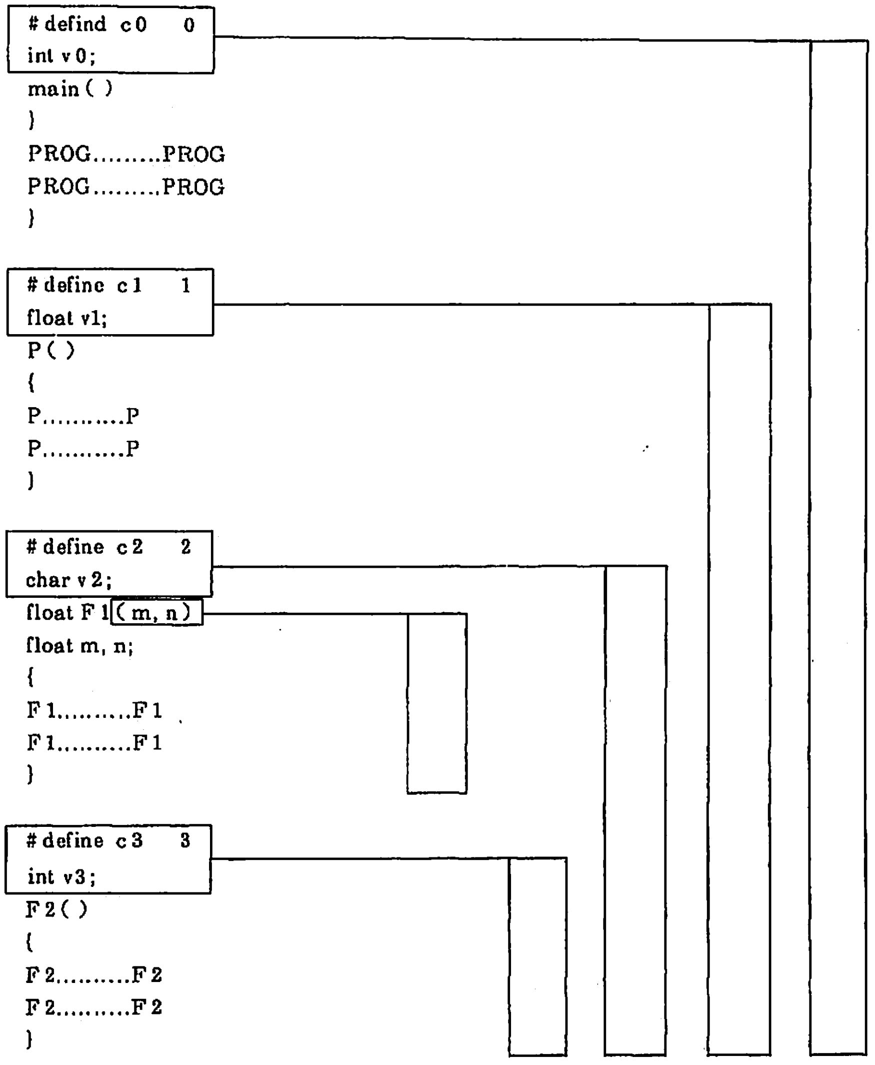
program PROG; Scope of variables



```
program PROG;

  const c 0=0;
  var v 0: integer;

    procedure P;
      const c1=1;
      var v1 :real;

        function F 1(m, n:real): real;
          const c 2 = 2;
          var v 2 :character;
          begin
              F 1..........F 1
              F 1..........F 1
          end;

        function F 2( ): real;
          const c 3=3;
          var v 3: integer;
          begin
              F 2..........F 2
              F 2..........F 2
          end;

        begin
          P.........P
          P.........P
        end;

  begin
  PROG........PROG
  PROG........PROG
  end.
```

B) The equivalent 'C' program

Scope of variables



```
# defind  c0    0
int v 0;
main ( )
}
PROG.........PROG
PROG.........PROG
}

# define  c1    1
float v1;
P( )
{
P...........P
P...........P
}

# define  c2    2
char v 2;
float F 1( m, n)
float m, n;
{
F 1.........F 1
F 1.........F 1
}

# define  c3    3
int v3;
F 2( )
{
F 2.........F 2
F 2.........F 2
}
```

strings, variable identifiers or function calls, with optional width-of-field specifications. For example:

```
write('length is', length:4:2, 'height is', height);
```

In 'C' a control string must be added to the equivalent functions scanf( ) and printf( ), to inform the 'C'-compiler explicitly of the types and number of arguments. The translator, with the above statement to translate, must produce:

```
printf("length is %4.2 fheight is %12f", length, height);
```

The translator checks that the variables "length" and "height" are valid variables, (that they have been declared AND are available at this point in the program), then finds their types, putting the appropriate symbols, d, c, s, f for integers, characters, strings and float variables respectively, having looked ahead, read and written any width-of-field details.

Notice the default width-of-field space allocation of 12 for variables. This corresponds to the action taken by a Pascal-compiler.

(8) Readln and writeln

The associated functions READLN and WRITELN also have to be translated. WRITELN is simply handled, by adding a final '¥n' in the control string of the printf statement.

A call is written by the translator into the translation memory to another function, readln ( ), which is automatically linked to the translated file before compiling. (This function is part of the translation program, not a predefined function of 'C's). This function carries out the same action as Pascal's empty "READLN;" statement.

(9) EOF and EOLN

Two more predefined functions in Pascal are EOF and EOLN. These are not available in 'C' and have to be separately made. These functions must then be compiled with the translated program before being run. EOF and EOLN have TRUE values (unity) if the pointer to the input data is pointing at the end of that date file, or the end of a line of it respectively. When for example EOF is found in the source program, the translator must write a call to the function _end ( ), (which is compiled with the final translation), specifically to look for EOF. The call looks like:

```
_end (OF_FILE)        [_end (OF_LINE) for case EOLN]
```

where OF_FILE and OF_LINE are simply constants included in the translated program automatically, (they must be written in the function main ( ) before any translation starts).

(10) Symbols

Various symbols are different in Pascal and 'C' but their meanings are the same. They need to be replaced where found. The following table lists them in both languages, together with their description:

DESCRIPTION	PASCAL	'C'
inclusive "or"	OR	||
"and"	AND	&&
real division and	DIV	/

truncating integer division		
modulus operator	MOD	%
assignment	:=	=
equality testing	=	==
inequality	<>	!=
complement	NOT	!
EOF check	EOF	_end (OF_FILE)
end-of-line check	EOLN	_end (OF_LINE)

(11) Various predefined functions

Pascal's predefined function are easily added to a translation program, and are listed below, but may vary from compiler to compiler:

ABS	GET	PAGE	SQR
ARCTAN	INTEGER	PRED	SQRT
BOOLEAN	INPUT	PUT	SUCC
CHAR	LN	READ	TEST
CHR	MAXINT	READLN	TEXT
COS	NEW	REAL	TRUE
DISPOSE	ODD	RESET	TRUNC
EOF	ORD	REWRITE	UNPACK
EOLN	OUTPUT	ROUND	WRITE
EXP	PACK	SIN	WRITELN
FALSE			

As an example of how some of these may be made, here are possible TRUNC and ROUND functions, used for converting real values to integers (before e. g. assignment to an integer variable).

(a) Example of action of TRUNC and ROUND:

	if	
	A:=3. 6	A:3. 4
ROUND (A) =	4	3
TRUNC (A) =	3	3

(b) Possible functions trunc and round in 'C':

```
trunc (px)
float*px;
{
return (*px) ;
}

round (px)
float*px;
```

```
{
float fl;

fl=*px+0. 5;
return (trunc(&fl));
}
```

Trunc takes advantage of the fact that if not otherwise declared, a 'C' function is assumed to return type int (or character). So returning a float will truncate any decimal part. Round simply adds 0. 5 and then truncates.

(C) Calls to trunc ( ) and round ( ) as they would be written by the translator to the translation once these functions are available would look like:

```
{                 {
float A;          float B;
....              ....
trunc (&A);       round (&B);
....              ....
}                 }
```

TRUE and FALSE may be included in the TRANSLATED main ( ) as:

```
#define    TRUE        1
#define    FALSE       0
```

(12) 'C's abbreviated operators

'C' has a couple of very useful abbreviations which Pascal has not got. These are shown below:

```
'C'                 PASCAL EQUIVALENT
-----               -----------------
a++                    a:=a+1
f(a++)                 begin
                          f(a);
                          a: = a+1
                       end

g(--a++)               begin
                          a:=a-1;
                          g(a);
                          a:=a+1
                       end

a+=3;                  a:=a+3;
```

etc.

SIMILARITIES BETWEEN PASCAL AND 'C'

Everything apart from those points in Differences, and also some points not covered by the translator constructed. For example, Pascal's "RECORD " is very similar to 'C's "struct ":

```
                    PASCAL                          'C'
                    ______                          ___

definiton:
__________

     type delivered =                         struct delivered {
                 record
                    color: character;               char color;
                    number: integer;                int number;
                    price: real;                    float price;
                 end;                               }

     var corn: delivered;               struct delivered corn;

calls:
______

   corn. number: = 123;                 corn. number = 123;
   read (corn.color);                   scanf ("%c", &corn. color);
   write (corn. price);                 printf ("%f", corn. price);
```

ARRAY TREATMENT

There are complicated and easy ways of writing Pascal programs using arrays. The two ways shown below for example are equivalent, but obviously the second is the translationally easier to deal with:

```
(1)                 type
                        morning hours = 1..12;
                        frequency     = morninghours;
                        table         = array [frequency] of real;

                    var
                        temperatures: table;


(2)                 var
                        temperatures: array [1..12] of real;
```

It is not a difficult task to check the grammar of the latter expression, and write the equivalent expression in 'C' putting the name of the array into the memory used for storing types of variable identifiers, and perhaps what type the array is designed to hold.

DIFFICULTIES AND INTERESTING POINTS

(1) EOF and EOLN

Pascal automatically puts EOF and EOLN true if the pointer to the input data is pointing to these, while keeping the pointer on them, Using getchar ( ) to check if the next character of the input is '￥n', '￥0' or EOF in the translator's equivalent function .end ( ), acting in the same way as Pascal's predefined functions, is only part of the answer. This is because getchar ( ), ('C's basic function for reading the next character of input), automatically moves the pointer it and scanf ( ) use forward to point to the next character, ready for the next call to either of these functions. Simply moving this pointer back again is also only part of the solution, because there is a double-check in getchar ( ) using a counter of the number of remaining characters of input. When the pointer points to EOF or '￥0', or when this counter reaches zero, the function getchar ( ) ceases to operate. So to keep 'C's pointer stationary while looking at the next character of input needs two statements counteracting these two automatic effects.

(2) Read

Character data in Pascal is written in stream with other data, with no intervening spaces. If a 'C' program uses scanf ("%c", &char_memory), the function scanf ( ) will skip any blanks and read only valid characters.

Thus 'C's data may be more clearly written, (assuming the scanf ( ) function is used in this way).

(3) Write/writeln

In this translator program, if no width-of-field is specified in a WRITE or WRITELN statement, an allowance of 12 character spaces is set by default. This is the same as Pascal's default, but 'C's default allowance is only 8 spaces usually.

(4) Translation difficulties

Perhaps the biggest difficulty in the translation is changing the nested structure of Pascal's subprograms into 'C's structure. A translator function is needed to write the first declaration lines of a block, (from the main program or from a sub-nested procedure or function), into a memory, then clear the file (into which all translated lines are written), look for any sub-programs, and write these into file if found. If there is more than one of these, the second would be written into the same file as the just-written one unless again a copy is made. So as sub-programs are translated they must be added into another memory, and file again cleared for possible further programs. When there are no more found, the BEGIN...END of the original program must be translated, and added to the original memory to its first few declaration lines. When THIS is done, it and any lower programs found in it, must be added together and returned to the calling function, (which itself may well be a subprogram).

This ensures that Pascal's procedures are written in grandmother-mother-daughter sequence in the final equivalent 'C' program, so that variables declared in mother procedures will be known to the daughters.

CONCLUSION

While Pascal and 'C' are very similar languages, it can be seen that there are occasions where a translator may be required between the two. Whereas grammar in the languages is very similar at the level of expressions and statements, (with one or two interesting differences), it is perhaps the overall structure of the two sets of programs which differ most, with accompanying differences in scopes of variables. These naturally give rise to the most difficulties in translating one to the other. In additon it is necessary to construct an equivalent set of available functions written in the target language which may be called, and function, compatibly with those available functions which exist in the object language.

REFERENCES

[1] K. Jensen and N. Wirth, *Pascal User Manual and Report*, 2nd ed., New York: Springer-Verlag, 1978.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, New Jersey: Prentice-Hall, 1978.