AN ABSTRACT OF THE THESIS OF

John A. Bertani for the degree of Master of Science in Computer Science presented on March 19, 1985.

Title: Achieving Portability Through Software Conversion.

Abstract approved: Redacted for Privacy

Theodore G. Lewis, Assoc. Professor

Conversion of software written for one machine or operating system to equivalent software for another machine or operating system is shown to be economically attractive using source-to-source translation. The features of an automatic converter are described using a Pascal-to-C translater as an example. Solutions to the problems of denesting procedures, converting data structures and types, converting control structures and operators, and converting semantics in one language into equivalent semantics in another language are proposed and evaluated in terms of an algorithmic translator. The results obtained through experience suggest that algorithmic translation from one language to another can yield 95 to 99 per cent conversion without human intervention, leading to significant improvements over other methods of software conversion.

# Achieving Portability Through Software Conversion

by

John A. Bertani

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed March 19, 1985

Commencement June 1985

APPROVED:

Redacted for Privacy

Assoc. Professor of Computer Science

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented March 19, 1985

Typed by Judith Sessions for John A. Bertani

## TABLE OF CONTENTS

## LIST OF FIGURES

FIGURE Page

# LIST OF TABLES

# Achieving Portability through Software Conversion

## Introduction

"Software is portable if it can, with reasonable effort, be made to run on computers other than the one for which it was originally written."
Brown (1977)

Software portability has become an important issue where micros, minis and mainframes are integrated into a single data processing environment. When micros, minis, and mainframe hardware is integrated it is necessary to either standardize operating systems and programming languages to minimize software development expenditures and maximize software compatibility or to find a way of achieving portability by a conversion process. In this paper we show how software conversion can be automatically achieved by source language to source language translation.

There are many reasons to consider source-to-source language translation as a means of software conversion. A switch to a new operating system; lack of a suitable compiler in the new environment; or programmers wish to change their primary language to take advantage of certain features in the new language.

Brian Ford (1983) presented a "wish list for easy transfer and sharing" of software which he considered necessary if numerical algorithms are to be converted to another environment. Peter Wallis (1982) also gave some recommendations for making software portable.

However, many programmers are faced with the problem of converting existing software which was written with a specific goal in mind, in a specific language and, most likely, under a deadline. More often than not, the issue of portability was not considered until after the software was written.

Programs to be converted to another environment but in the same language often run into problems due to differences in dialects. Although many attempts have been made to standardize programming languages, enhanced versions will always be found and because of the enhancements, they will proliferate. Enhancements are made with the intention of making the programmers job easier but from a portability point of view, make the conversion programmer's job more difficult.

Wolberg (1983) states that the decision to convert is basically an economic decision. The three alternatives he cites are to convert the software, replace the software, or discard the software. He compared the cost of converting software to the cost of reprogramming Wolberg (1978). Walston and Felix (1977) analyzed a database of 60 software development projects ranging in size from 4000 to

467,000 lines and came up with the following equation for new system development effort of

$$\text{Effort} = 5.2 * (\text{\# of Lines})^{0.91}$$

Wolberg assumed that reprogramming required approximately one-half the effort of new system development and obtained the following relationship:

$$\text{Effort}_r = 2.6 * (\text{\# of Lines})^{0.91}$$

for reprogramming effort and

$$\text{Duration}_r = 3.3 * (\text{\# of Lines})^{0.36}$$

for project duration.

Wolberg used a database of 9 completed conversion projects to obtain the conversion effort equation:

$$\text{Effort}_c = 7.14 * (\text{\# of Lines})^{0.47}$$

and a database of 31 completed conversion projects for the duration equation:

$$\text{Duration}_c = 4.1 * (\text{\# of Lines})^{0.22}$$

where for all equations, Effort is in person-months and Duration is in months. These formulas were obtained for projects where automated conversion tools were used to partially translate from one system to another (equivalent) system.

From Table I we can conclude that the larger the system to be converted the higher the ratio of reprogramming to conversion effort. However, the conclusion depends on having an automated "converter" which performs at least part of the translation with minimal effort.

## Steps in Converting

When converting software by source-to-source translation there are two steps:

1. Translate the original source language into the target source language. This includes translation of control structures, operators, declarations, and libraries.
2. Testing and debugging.

Testing and debugging is a normal phase of every software development project and should progress fairly rapidly in a conversion project because of the availability of a working version of the original source code. The main interest of this paper is the translation process which can be further broken down into two steps:

1. Automatic translation
2. Manual translation

Table I. Results of Wolberg Study

|  | Effort Ratio | Duration Ratio |
|---|---|---|
| Lines | Er/Ec | Dr/Dc |
| 30000 | 1.63 | 1.29 |
| 100000 | 2.76 | 1.15 |
| 300000 | 4.48 | 1.78 |
| 1000000 | 7.61 | 2.10 |

Automatic translation implies algorithmic conversion. Algorithmic conversion is any conversion technique that follows an outline, in a step-by-step, systematic fashion. High level languages are sufficiently complex that most automatic converters cannot translate at 100 per cent efficiency. Here, efficiency is defined as the percentage target source code that needs no modification, syntactic or semantic, in order to produce identical results when implemented in the target environment. As can be seen from the work of Wolberg, translation efficiency of 95 to 99 per cent becomes a better economic choice as the size of the original source code program increases.

## Conversion Difficulty

Wolberg classified conversions into three categories and gave them an ease-of-conversion rating (Table II). The following definitions are used in Table II:

1. Intralanguage: Target language is a different version of the same source language (e.g. UCSD Pascal to standard Pascal).
2. Interlanguage: Target language is different from the source (e.g. Pascal to C)
3. Same Compiler: The source and target both use the same version of a given language (e.g. a prettyprinter or an optimizer).

Table II. Wolberg's Rating

| | Source | | Target | | |
|---|---|---|---|---|---|
| Type | Lang. | Vers. | Lang. | Vers. | Ease of Conversion |
| Intra-Language | L1 | V1 | L1 | V2 | Easy to difficult |
| Inter-Language | L1 | | L2 | | Difficult to very difficult |
| Same compiler | L1 | V1 | L1 | V1 | Usually easy |

In Table II, 'easy' means algorithmic conversion results in close to 100 per cent effectiveness. 'Difficult' implies a more complex task; much of the conversion can be done algorithmically but visual examination of the results is needed to complete the translation manually. 'Very difficult' means little help can be derived from algorithmic translation.

Many automatic interlanguage converters claim a translation percentage greater than 95 per cent. Even so, this may mean that some original programs translate 100 per cent while others may have semantic errors introduced by the translation. As an example, consider the case of translating Pascal's READLN into an "equivalent" FSCANF of C.

```
readln (file, argument1, argument2);
```

where 'file' is the file being scanned and 'argumentX' is the variable being assigned a value. If 'argument1' is an integer, 'argument2' a character, and READLN's are translated into FSCANF's, the resulting C code might be:

```
fscanf (fileptr,"%d%c", &argument1, &argument2);
```

This is an acceptable translation only if there are two values of data in the record of the input file. But consider the case of reading a string of characters containing imbedded blanks (as in UCSD Pascal).

```
        readln (file, astring);
```

Given the translation criteria above, this would translate into:

```
        fscanf (fileptr, "%s", astring);
```

In Pascal READLN scans the input string until an end-of-line marker is reached, but in C, FSCANF would stop at the first blank, tab or end-of-line character. An appropriate translation would be:

```
        fgets (astring, DIMENSION_OF_STRING, fileptr);
        astring[strlen(astring)-1] = 0;
```

This type of problem can be handled as a special case by an automatic converter. However, translations are generally full of special cases and adding code to detect them would only increase the code size and the execution time. For many of the special cases it might be better to manually change the code after visual inspection.

## Design of an Automatic Conversion Tool

The design of an automatic conversion tool can be accomplished by examining the basic components of the source and target languages. Of primary importance are the data types and declarations, the instructions and operators, and the support libraries. Of lesser importance is the data storage methods associated with each language.

A converter is similar to a compiler in function: to convert programs written in a source language to an equivalent program written in another (target) language. However, a converter is different from a compiler in many respects.

A typical compiler might consist of the following parts:

1. Lexical analyzer
2. Parser
   a. syntactic analyzer
   b. symbol table
   c. parse tree
3. Semantic Analyzer
4. Code Generator

A converter normally consists of the following:

1. Preprocessor
2. Lexical Analyzer
3. Parser
   a. symbol table
   b. parse tree
4. Code Generator

A preprocessor is a program used to prepare the source code for translation. A converter does not need a syntactic analyzer because the source language, having been compiled, is free of syntax errors. For the same reason, the converter does not need a semantic analyzer. A parse tree is necessary if the languages are different in structure and a line-by-line translation is not possible.

For languages with similar constructs, the target language can be generated directly from the input tokens passed to it by the lexical analyzer. The more common case encountered in conversion is a mix of similar and dissimilar structures. In this case the preprocessor is necessary to simplify the converter.

As an example, consider a converter for translating Pascal source language programs into some other target language. A Pascal program consists of a sequence of declarations followed by a single compound statement. The declarations consist of:

1. labels
2. constants
3. types
4. variables
5. procedures and functions

Each procedure or function may contain its own set of declarations in addition to its own single compound statement.

Nesting is achieved by declaring procedures and functions inside the program and also inside other procedures and functions. Most Pascal compilers use a recursive descent parsing algorithm to make one pass over the source code, translating on a line-by-line basis. However, this simple approach does not work for a converter if the target language does not support nested procedures. Instead, a preprocessor must de-nest the Pascal procedures before they can be translated by syntactic analysis.

The preprocessor examines the source code with the following objectives in mind:

1. de-nest the Pascal program
2. create a global symbol table (for rewriting the declarations)
3. move non-local, non-global identifiers to the global symbol table (including all types and constants).
4. simplify or rephrase dissimilar statements and operations (such as WITH and set operations).
5. create a pseudo Pascal program in a format acceptable to the translator.

The preprocessor uses a stack to deal with de-nesting of procedures and functions. Since the body of the program, procedures, and functions is of no concern to the preprocessor (except for simplifying or rephrasing) only data declarations need be stored. A typical variable declaration and the structure created from it are shown in Figure 1.

When the preprocessor encounters a nested function or procedure, it places the current structure on a stack, creates a new structure as shown in Figure 2, and analyzes the declaration section of the nested procedure or function. It continues to push structures onto the stack and create new ones until it reaches the instruction list of the procedure or function it is analyzing.

After analyzing the nested procedure/function, the entire procedure/function can be written to an intermediate file, the current structure discarded, and the stack popped to resume analysis. (In practice this is done using recursion.) Following this procedure, the entire Pascal program is denested.

A side effect of the de-nesting algorithm is that non-local, non-global identifiers are left "homeless". The preprocessor adds these declarations to the declarations kept in the global data structure. To avoid conflicts in naming of identifiers, the preprocessor can prepend a prefix (based on the procedure or function in which it was declared) to the identifier to distinguish it from other like-named identifiers.

A second pass over the source code converts the intermediate file to C one line at a time. In creating the pseudo Pascal program for the translator portion of

Declaration: list : array [1..100] of integer;

Structure:

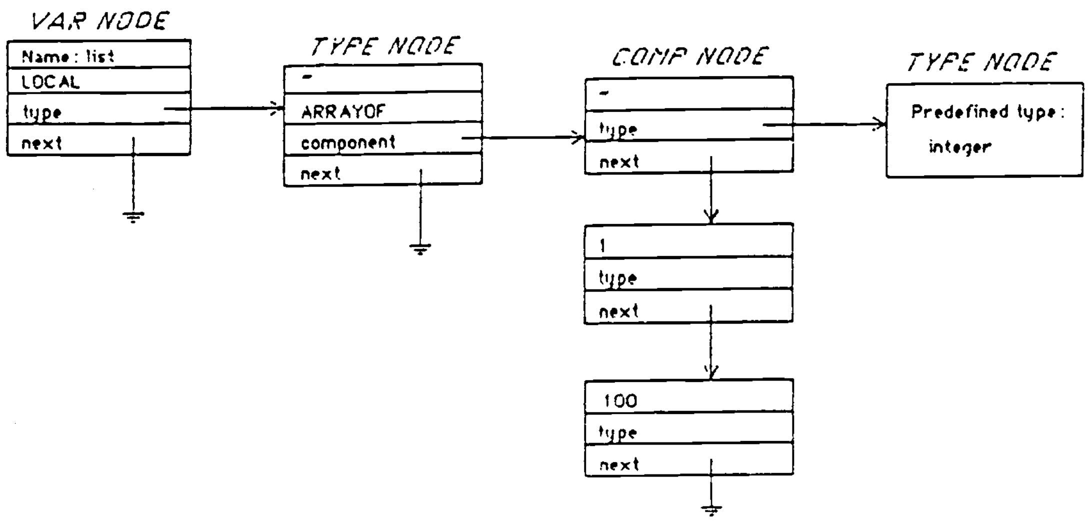

FIGURE 1. Variable Declaration Example

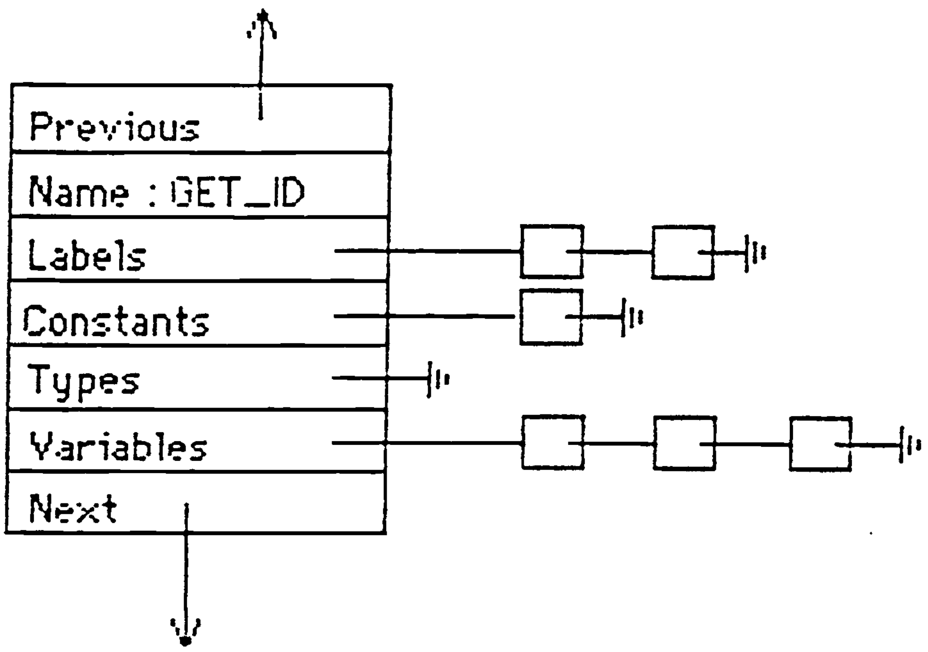

FIGURE 2. Procedure-Function Node

the conversion process, the preprocessor creates a file in which everything is declared before it is used (except pointers). As the translator processes each line it creates a symbol table entry for each identifier and keeps a table of attributes so that at any time information can be retrieved about each identifier. The data structure used in this case is an array of structures whose contents are outlined in Table III.

As the translator processes a procedure or function it appends parameters and local declarations to the attribute table. When it is ready to process the statements, all the information necessary to generate the target code is available. Once it has finished processing the procedure or function it removes the parameters and local declarations from the symbol table and moves on to the next procedure/function or main body.

## Conversion of Data Structures

Ease of conversion of data structures is based on the similarity of the two languages involved. Obviously, the more restricted the target language is in terms of available data types, the more difficult the conversion process will be. Consider the conversion of Pascal both to another dialect of Pascal and to C. Table IV shows the data types involved in intralanguage conversion of data types.

Table III. Attribute Table Layout

| itemtype | | var | name | supertype | lower | upper |
|---|---|---|---|---|---|---|
| 1 | boolean | 1st var | name ptr | see #1 | see #2 | |
| 2 | char | 1st var | name ptr | see #1 | see #2 | |
| 3 | integer | 1st var | name ptr | see #1 | see #2 | |
| 4 | real | 1st var | name ptr | see #1 | see #2 | |
| 5 | subrange | 1st var | name ptr | see #1 | see #2 | points to array dim |
| 6 | string | 1st var | name ptr | see #1 | see #2 | string dim |
| 7 | pointer | 1st var | name ptr | see #1 | see #2 | type pointer |
| 8 | long | 1st var | name ptr | see #1 | see #2 | size of long |
| 9 | packed | 1st var | name ptr | see #1 | see #2 | type pointer |
| 10 | array | 1st var | name ptr | see #1 | see #2 | ptr to 1st dim |
| 11 | record | 1st var | name ptr | see #1 | see #2 | 1st field ptr |
| 12 | set | 1st var | name ptr | see #1 | see #2 | type ptr |
| 13 | file | 1st var | name ptr | see #1 | see #2 | type ptr (0 if no structure) |
| 14 | array dim | pointer to sub-range; 0 if const | | ptr to next dim (last points to type) | lower if const | upper if const |
| 15 | var | next var of this type | name ptr | type pointer | 1 if call by reference; 0 if call by value | |
| 16 | const | integ-erized value if char int or boolean | name ptr | see #3 | | |
| 17 | type ptr | 1st var | name ptr | see #1 | see #2 | pointer to actual type |
| 18 | union | | | see #1 | 1st field | |
| 19 | procedure | | name ptr | see #1 | 1st formal parameter | |
| 20 | function | | name ptr | see #1 | 1st formal parameter | pointer to return type |

Note #1: This field is zero if not a type within a type like records, files, sets, and arrays. Otherwise it is a pointer to the type's superstructure.

Note #2: This is the next field pointer. It points to the next type field within a record.

Note #3: This field is the constant type. The types are:

```
          1  char
          2  float
          3  integer
          4  string
          5  long
```

Comments on some of the itemtypes listed above:

Type 17 records are dummy type records. UPPER of a type 17 record points to its actual type. This type of record is used for several reasons, the most common being where say BOOL is declared as type BOOLEAN and BOO is declared as type BOOL. Then BOOL gets a type 1 record in the attribute table and BOO gets a type 17. Now each record can point to the linked list of variables that were declared having type BOO or BOOL.

Arrays have a type 10 record in the attribute table. UPPER points to a type 14 record which is an array dimension record. If there is more than one dimension, the SUPERTYP field in the type 14 record points to the next type 14 record. The SUPERTYP field in the last type 14 record points to the record that contains the type of the array. Type 14 records have a zero in their VAR field if the lower and upper bounds of the dimension are in the LOWER and UPPER fields. Otherwise VAR points to a subrange record.

Subrange records point to array dimension records and the lower and upper value for the subrange are in the LOWER and UPPER fields of the array dimension record.

Table IV. Intralanguage Conversion of Data Types

| Pascal MT+ | UCSD Pascal |
|---|---|
| BYTE | 0..255 |
| WORD | 0..65535 |
| LONGINT | INTEGER [X] |

| MS Pascal | UCSD Pascal |
|---|---|
| WORD | 0..MAXWORD |
| REAL4 | (1) |
| REAL8 | (1) |
| INTEGER4 | INTEGER [X] |
| STRING (N) | PACKED ARRAY [1..N] OF CHAR |
| LSTRING (N) | STRING [N] |
| SUPER ARRAYS | ARRAYS |
| ADR OF | (2) |
| ADS OF | (2) |

(1) since there are no equivalent types in UCSD Pascal, the programmer making the conversion must decide whether to replace with standard real numbers or create a library of extended real operations.

(2) Since there is no eqivalent in UCSD Pascal, the programmer must decide on the course of action to take with the address-of types.

Table V shows the interlanguage conversion of the data types.

## Conversion of Control Structures and Operators

The conversion of control structures can be broken down into two obvious groups. First and easiest to translate are the constructs in which a version exists in both the source language and the target language with slight syntactic differences. Table VI shows the mapping between Pascal and C control structures.

The other group is the control structures in which there are no equivalent constructs in the target language. An example of this is Pascal's WITH statement. The algorithm to translate WITH can take three forms. The first is to prepend the with variable to the appropriate fields contained within the the WITH statement list. The second is similar to the first but takes advantage of C's macro preprocessor. Instead of prepending the WITH variable to each appropriate field element, it is assigned to a local variable (created by the translator) using the '#define' mechanism. The third algorithm is similar to the second but does an actual assignment rather than a macro substitution. Table VII shows how WITH statements are translated using each algorithm.

Operators can also be broken into two groups. If there is a counterpart in the target language then the conversion

Table V. Interlanguage Conversion of Data Types

| UCSD Pascal | C |
|---|---|
| BOOLEAN | INT |
| INTEGER | INT |
| INTEGER [N] | REAL or CHAR (1) or LONG |
| REAL | FLOAT (or DOUBLE) |
| CHAR | CHAR |
| STRING | CHAR ___[X] |
| ARRAY (some type) | type ___[X] |
| RECORD | STRUCT |
| VARIANT RECORD | UNION |
| ENUMERATED TYPE | INT (for versions void of enumerated types) |
| SETS | bits of CHAR array |
| FILES | STRUCT containing file element type and file pointer |

(1) The programmer can convert the UCSD long integers to ASCII strings and use some well-known algorithms to simulate long arithmetic.

Table VI. Control Structure Mapping, Pascal to C

```
IF <condition> THEN            if (<condition>)
    <statement-list>               <statement-list>
ELSE                           else
    <statement-list>               <statement-list>

FOR lcv:=initial TO final DO for (lcv = initial; lcv <= final; lcv++)
    <statement-list>             <statement-list>

REPEAT                         do
    <statement-list>                 <statement-list>
UNTIL <condition> ;            while (!<condition>) ;

WHILE <condition> DO           while (<condition>)
    <statement-list>                 <statement-list>

CASE <expression> OF           switch (<expression>) {
    <constant>: <stat-list>    case <constant>: <stat-list>
     .                                          break;
     .                              .
     .                              .
END                              }
```

Table VII. Three Translations of Pascal's WITH statement

```
WITH a_record_struct DO BEGIN       /* with a_record_struct do */
    field1 := _______;              a_record_struct.field1 = ______;
    field2 := _______;              a_record_struct.field2 = ______;
END;

WITH a_record_struct DO BEGIN       #define  withp0  a_record_struct
    field1 := _______;                  {
    field2 := _______;                  withp0.field1 = _______;
END                                     withp0.field2 = _______;
                                        }
                                    #undef withp0

WITH a_record_struct DO BEGIN       withp0 = &a_record_struct;
    field1 := _______;                  {
    field2 := _______;                  withp0->field1 = _______;
END;                                    withp0->field2 = _______;
                                        }
```

is easy and similar to a word processing 'change' operation. If there is no counterpart, then the operations must be simulated. Operators that do not have a target language complement can be simulated with library calls to routines performing the same operation.The most obvious of this group in the Pascal-to-C translator is the set operators. Table VIII shows the Pascal set operators and the equivalent C library routine.

The code generation format for statement lists is similar to the recursive algorithm found in many compilers. From Table VI we see that the translation of REPEAT-UNTIL calls for the translation of a statement list. Figure 3 shows the part of a procedure (called Gen_Statement) that translates this construct.

## Conversion of Support Libraries

If the support library of the source language cannot interface with the target language, then the support routines will have to be simulated with a new support library. Most often the same names can be used in the new support library with the same arguments unless there happens to be a name conflict in the target language. If the same names are used in both libraries then the statements containing these calls can be processed without many changes. It is then only necessary to write the support library routines for the target language.

Table VIII. Pascal Set statements, C equivalent statements

| Pascal Operation | C Function |
|---|---|
| `set1 := set2 + set3;` | `set1 = p2c_s_union (set2, set3);` |
| `set1 := set2 - set3;` | `set1 = p2c_s_diff (set2, set3);` |
| `set1 := set2 * set3;` | `set1 = p2c_s_intrs (set2, set3);` |
| `set1 := [1,2,3..6,7];` | `set1 = p2c_set (1,2,3,t_o,6,7,e_n_d);` |
| `if (x in set1) then` ..... | `if (p2c_s_in (x,set1))` ..... |
| `if (set1 = set2) then` ..... | `if (p2c_s_eql (set1,set2))` ..... |

```
Gen_statement();
{
      .
      .
      .
      switch (statement_type) {
            .
            .
            .
           case REPEAT:
                  fprintf (out, "do {\n");
                - Gen_Statement();
                  get_token();
                  if (strcmp (token,"until") != 0)
                      Error ();
                  fprintf (out, "while (!");
                  Gen_Expression();
                  eoln();
                  break;
            .
            .
            .
            .
            .
      }  /*  end of switch  */
}
```

---

**FIGURE 3.** REPEAT-UNTIL of Gen_statement()

In many cases, the target language support library contains routines that are semantically equivalent to the source language support routines. The translator can then translate the procedure/function call to the equivalent call in the target language. (Often this entails a rearrangement of the arguments in the parameter list). This would eliminate the overhead of two procedure calls in the compiled version of the target language. Table IX shows the mapping of some Pascal support library routines to the C support library.

## Comparison of Pascal, C and translated C

One of the criticisms of high level code that comes out of an A-to-B translator is that it is an A program with B syntax. This is true! The translator is not an intelligent program. It translates on a token-by-token basis and does not do any context analysis. But, a compiler for the B language only recognizes B syntax and will generate code according to what it sees. The code generated for A and the code generated for B are different as the following examples of a Pascal-to-C translation illustrate. The first example is the benchmark Sieve of Erastothenes. The second example is a Pascal WITH statement of which there is no equivalent in C. Both examples were run in the UNIX operating system using the pc compiler for Pascal and the cc compiler for C.

Table IX. Pascal Support Library mapped to C Support Library (some Pascal routines are from UCSD Pascal)

| Pascal Routine | C Function |
|---|---|
| assign(file,name); reset(file) | file = fopen(name,"r") |
| blockread (filename, buffer, count) | fread (buffer, size, count, filename.fileptr) |
| blockwrite (filename, buffer, count) | fwrite (buffer, size, count, filename.fileptr) |
| concat (s1,s2,s3,...) | sprintf(temp,format,s1,s2,s3....) |
| dispose (ptr) | free (ptr) |
| exit (program) | exit (0) |
| exit (routine_name) | longjmp (env, value) |
| get (filename) | read (filenum, buffer, length) |
| halt | _exit () |
| new | malloc (size) |
| put (filename) | write (filenum, buffer, length) |
| read[ln] ([filename], variable-list) | fscanf (fileptr, format, variable-list) |
| reset (filename, [,external_file]) | reset (filenum, external_file) |
| rewrite (filename, [,external_file]) | rewrite (filenum, external_file) |
| write[ln] ([filename,] variable_list) | fprintf (fileptr, format, variable_list) |

Example 1 - Sieve of Erastothenes

C version - from Kern

```
    #include <stdio.h>
    #define  SIZE   8190
    #define  FALSE     0
    #define  TRUE      1
    #define  NTIMES   10
    char    flag[SIZE + 1];

    main()
    {
    int      i,j,k,count,prime;

    printf ("10 iterations: ");

    for (i = 1; i <= NTIMES; i++) {
            count = 0;
            for (j = 0; j <= SIZE; j++)
                    flaj[j] = TRUE;
            for (j = 0; j <= SIZE; j++) {
                    if (flag[j]) {
                            prime = j + j + 3;
                            for (k = j + prime; k <=
                             SIZE; k += prime)
                                  flag[k] = FALSE;
                            count++;
                    }
            }
   }
   printf ("%d primes.\n", count);
   exit(0);
   }
```

Pascal Version - Public Domain Software

```
   program sieve;
   const
       SIZE = 8190;
       NTIMES = 10;

   var
       i,j,k,count,prime : integer;
       flag : array [1..SIZE] of boolean;
```

```
begin
    write ('10 iterations: ');
    for i := 1 to NTIMES do begin
        count := 0;
        for j := 1 to SIZE do
            flag[j] := true;
        for j := 1 to SIZE do begin
            if flag[j] then begin
                prime := j + j + 3;
                k := j + prime;
                while k <= SIZE do begin
                    flag[k] := false;
                    k := k + prime;
                end;
                count := count + 1;
            end;
        end;
    end;
    writeln (count, ' primes.');
end.
```

Pascal-to-C Translator Version

```
#include "stdio.h"
#define TRUE 1
#define FALSE 0
#define SIZE 8190
#define NTIMES 10

int         flag [8192] ;
int         i ,
            j ,
            k ,
            prime ,
            count ;

main()
{
   printf("%s\n","10 iterations: ");

   for (i = 1; i <= NTIMES; i++) {
         count = 0;
         for (j = 0; j <= SIZE; j++)
            flag[j] = TRUE;
         for (j = 0; j <= SIZE; j++)
            if (flag[j]) {
                  prime = j + j + 3;
                  k = j + prime;
                  while (k <= SIZE) {
                        flag[k] = FALSE;
                        k = k + prime;
                        }
                        count = count + 1;
                    }
            }
            printf("%d%s\n",count," primes.");
        }
```

From the two C programs, it can be seen that there are only a few differences (other than Initialization and I/O) to be noted:

```
        for (k = j + prime; k <= size; k += prime)
                flag[k] = false;
```

versus

```
        k = j + prime;
        while (k <= size) {
                flag[k] = false;
                k = k + prime;
        }
```

Clearly, the two pieces of code perform the same function and the assembly language versions show that therets

is only a slight difference in the code produced by the compiler for each C version.

```
   addl3   -12(fp),-4(fp),r0         addl3   -12(fp),-4(fp),r0
   movl    r0,-8(fp)                 movl    r0,-8(fp)L26:
                                     L26:
   cmpl    -8(fp),$100               cmpl    -8(fp),$100
   jgtr    L25                       jgtr    L25
   movl    -8(fp),r0                 movl    -8(fp),r0
   clrl    _flag[r0]                 clrl    _flag[r0]
L24:                                 addl3   -12(fp),-8(fp),r0
   addl2   -12(fp),-8(fp)            movl    r0,-8(fp)
   jbr     L26                       jbr     L26
L25:                                 L25:
```

However if a comparison is made of the code from the C compiler to that from the Pascal compiler below it shows quite a difference.

```
addl3   _prime,_j,r0
movl    r0,_k
L6:
movl    _k,r0
cvtbl   $100,r1
cmpl    r0,r1
jgtr    L7
moval   _flag,r0
subl3   $1,_k,r1
clrb    (r0)[r1]
addl3   _prime,_k,r0
movl    r0,_k
jbr     L6
L7:
```

Example 2 - Pascal WITH statement

Here the purpose is to show the difference in the code from the two compilers.

Pascal code

```
with aptr^.next^ do begin
    var1 := 1;
    var2 := 2.0;
    next := nil;
end;
```

Pascal-to-C translator Version

```
withp0 = aptr->next;
    {
    withp0->var1 = 1;
    withp0->var2 = 2.0;
    withp0->next = NULL;
    }
```

Assembly language versions -

```
        movl    *$_aptr,sp                 movl    _aptr,r0
        movl    12(r0),r11                 movl    12(r0),-4(fp)
        movl    $1,(r11)                   movl    $1,*-4(fp)
        .data                              .data
        .align 2                           .align 2
L6:                                L24:
        .double 0d2.0000e+00               .double 0d2.0000e+00
        .text                              .text
        movd    L6,4(r11)                  movl    -4(fp),r0
        clrl    12(r11)                    movd    L24,4(r0)
                                           movl    -4(fp),r0
                                           clrl    12(r0)
```

An A program with B syntax does have at least one advantage. Programmers converting their high level code from A to B may not be familiar with the target language. A gradual change from language A to language B would be the preferred course. However, because of other conditions, this is not possible. A translator producing A-flavored code in language B is an acceptable alternative. The programmer will recognize the syntax more easily and, as he becomes familiar with the new language, can phase out the old language's syntax.

## Test Results

The Pascal-to-C translator was used to convert approximately 24,000 lines of Pascal source code to C. Where non-standard Pascal was used, manual file preparation was done before passing the source code through the automatic preprocessor. An example of manual file preparation is MS Pascal's LSTRING super type which uses parentheses to declare maximum length. The efficiency of translation was broken down as follows:

1. 99% - syntactically, semantically correct
2. 1% - semantically incorrect
3. 0% - no attempt at translation

In this particular project, there were no instances of non-translatable code. The 1% category consisted of the following problems:

1. Some of the procedures and functions required the use of the address operator on strings and records passed as parameters. E.g.

   ```
   proc_call (arg1, addr(arg2));
   ```

   where arg2 is a record and ADDR is an operator that computes the address of its argument. C does not allow a structure to be passed as a parameter, only a pointer to a structure. The ADDR function is not given a special case status hence the translator converts record parameters to pointer parameters but does not drop the ADDR operator. A C string variable name is already the address of the first element and thus does not need the address computed.

2. Different return value for the POS function. Pascal returns 0 when the substring can not be found in the

target string. The C equivalent function returns -1. All references to POS and to any variable assigned the return value of POS needed to be checked in both subscript and conditional expressions. The C POS function was later changed to return a value of zero. This worked for translated strings because the translator subtracted 1 from all string subscripts (to account for the difference in starting subscripts).

3. The SIZEOF operator in Pascal accepted type names as arguments. The conversion process simply passed the argument through to the output file. The C SIZEOF operator would not recognize a Pascal type that does not have a C equivalent. E.g. type anarray = array [0..100] of integer;

4. RETURN statements inside functions. When the translator converted functions, it generated a RETURN statement if it encountered the function name on the left hand side of an assignment statement. In cases where the function value was given a default value before any processing was done an error was introduced into the file. Because this was foreseen, a comment was generated indicating the possible error.

5. Intrinsic functions. The majority of dialects of Pascal have their own intrinsic functions. Many of

these functions have equivalents in C or could be easily rewritten in C. But for those in which no suitable substitute could be found a stub was generated and flagged.

6. Operating system dependent code.

## Concluding Remarks

Wolberg sees program enhancement as a "natural activity within the realm of a conversion project". That is, enhancing the maintainability and the portability may be desired effects. If portability development is undertaken after the software development stage, a desired side effect of the conversion process might be the standardization of the software so that it may be ported to another environment in the future without too much extra work. This standardization can be incorporated in the automatic conversion tool such that the high level target language is as standard as possible. Thus, if the program is moved to another environment using the same language, the source language will be very close to being a subset of the target language. If it is moved to an environment in a new language, the standard flavor of the code will make the design of another automatic converter an easier task.

We have shown by example, how a Pascal-to-C translator works and that such a translator has been implemented and used for conversions by the authors. The techniques discussed can be generalized to other translators. We predict that source-code-to-source-code translation will become an increasingly important method of software conversion.

## BIBLIOGRAPHY

Brown, P.J. 1977. Software Portability. Cambridge University Press. Cambridge, England.

Ford, B. 1983. "Software Transfer and Sharing," Programming for Software Sharing. Reidel Publishing Company. Dordrecht, Holland.

Jensen, K. and Wirth, N. 1974. The Pascal User Manual and Report. Springer-Verlag.

Kern, C.O. 1983. "Five Compilers for CP/M-80." Byte Vol. 8, No. 8.

Kernighan, B.W. and Ritchie, D.M. 1978. The C Programming Language. Prentice-Hall, N. J.

Oliver, P. 1979. "Software Conversion and Benchmarking." Software World, Vol. 10, No. 3, pp. 2-11.

Peterson, J.L. 1984 "Translating Pascal into C," IEEE Software, Vol. 1, No. 3, pp. 82-86.

Wallis P.J.L. 1982 Portable Programming. John Wiley & Sons. New York, NY.

Walston, C.E. and Felix, C.P. 1977. "A Method of Programming
Measurement and Estimation," IBM Systems Journal. Vol. 17, No. 1, pp. 54-73.

Wolberg, J.R. 1978. "Comparing the Cost of Software Conversion to the Cost of Reprogramming." SIGPLAN Notices, Vol. 16, No. 4, pp. 49-59.

Wolberg, J.R. 1983. Conversion of Computer Software. Prentice-Hall, Englewood Cliffs, NJ.