An APL Compiler

Tilman P. Otto
Paul-Martin-Ufer 13
68163 Mannheim, Germany
Tilman@Otto.com

Abstract

Even if APL is the best-suited programming language for multi-dimensional data, nowadays computer applications additionally require complex graphical user interfaces, internet and database access. Combining software written in C, C++ or Java with interpreted APL programs is difficult. A homogeneous solution has been found by automatically converting APL programs into native C code. A complete APL2 like system including interpreter and session manager has been implemented in ISO C from scratch only based on the standard C library. The system is the property of the author and not commercially available yet. It has been successfully compiled on several operating systems. The built in system call `APL2C` allows one to compile any APL function including all referenced functions or operators within the workspace into native C code and completely removes the interpreter using direct calls to the C coded APL primitives. Only obvious restrictions (no runtime execution of character arrays or dynamic creation of functions via `FX`) apply. In addition, a makefile is created to enable the simple build of standalone executable files. The C files, generated by `APL2C`, can be easily mixed with other C/C++ source files and compiled on any platform provided that the required library for the APL primitives is available.

Introduction

In the course of the nearly 40 years of APL[1] history, one of the main problems was the missing possibility to create real executable files that could be easily spread to users. The idea of developing an APL compiler dates back to the middle 80ies [2,3,4]. The APEX project [5] aims at an analysis and complete understanding of the APL program data flow to be able to automatically generate highly optimized native C code. Given the complex functionality of APL with its primitive functions, operators and nested arrays, this turns out to be a very difficult task. The approach presented in this paper is a pragmatic rather than demanding one. The compiler converts APL functions into C code, replacing the interpreter by direct calls to the C coded APL primitives.

The APL2C System

A complete APL system has been developed by the author since 1991. It comprises an interpreter, a session manager and a simple function editor to define functions and operators. It supports all standard APL primitives (functions and operators), nested arrays, selective and vector assignments, whereas complex numbers are not supported. The system was implemented from scratch in ISO

```
token("0xE4")  char("0x8C")  (#)     /* quad
token("0xE5")  char("0x82")  Encode  /* up tack
token("0xE6")  char("0xB1")  Circle  /* circle
token("0xE7")  char("0xBD")  Shape   /* rho
token("0xE8")  char("0x97")  Max     /* up stile
token("0xE9")  char("0x87")  Drop    /* down arrow
```

Figure 1: The APL2C system allows to define keywords in a user definable ASCII file as a replacement of APL special symbols (e.g. the keyword "Drop" replaces the "down arrow" character). With the second column, every single character of a special APL font can be assigned to its corresponding internal token. This allows adapting the system for different APL fonts.

Standard C [6]; sole basis were calls to the standard C library. It was successfully compiled on several operating systems such as Windows 95/98/NT, Linux, SGI Irix, IBM AIX and Sun Solaris. Two different user interfaces are available: A text-based interface using VT100 terminal emulation (for unix platforms) and a graphical user interface for Windows, written in C++ using the Microsoft Foundation Classes (MFC), see figure 2. For the text-based interface it is possible to define keywords for every APL symbol in an external ASCII file (see figure 1), whereas the Windows interface allows to enter APL symbols using a toolbar or user definable hot key combinations. The system is able to store and load workspaces and the most important system variables and functions are available (e.g. *IO*, *CT*, *LC*, *FX*, *CR*, *NL*, etc.). Within a workspace, the user may define niladic, monadic and dyadic user functions as well as monadic and dyadic operators. In case of a run-time error or a user interrupt, the interpreter can be stopped after the execution of any line of APL source code. An execution stack was implemented and the system command "*)SI*" is used to view pending functions. With a call to the "*)RESET*" function, all pending functions are cleared from the stack. With the specification of an argument the given number of pending functions can be removed from the stack, e.g. "*)RESET 3*". In the current implementation, a workspace cannot be saved as long as there are still functions pending. With the command " *LC*" the execution of a pending function can be continued, or, with " ", the complete list of pending functions for the last submitted APL command line can be removed from the stack.

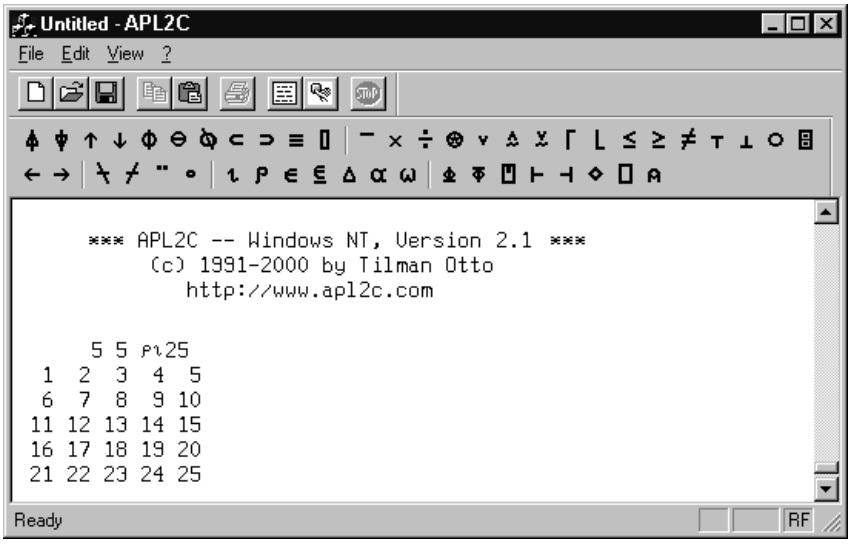

Figure 2: Screen snapshot of the APL2C system for Windows NT. The special APL symbols can be entered by means of the toolbar or a user definable hotkey combination.

```
typedef struct
{
      long           length;   // total length in bytes
      unsigned char  type;     // SIMPLE or NESTED
      unsigned char  rank;     // Dimension
      unsigned char  eltype;   // INTG, CHAR, REAL, …,PNTR
      unsigned char  reserved; // unused
      long           number;   // total number of elements
      long           dim[1];   // dimension(s), dep. on rank
      //             ...       // Data (dynamic)
} aplarray;
```

Figure 3: The definition of an APL array in C. The field ‘dim[]’ has, in fact, ‘rank’ elements and the array’s data directly follow the last ‘dim’ element.

Implementation

The implementation of a complete APL system is not that easy. A powerful low-level programming language is required to implement the interpreter and all APL primitives. What is furthermore needed is a high portability of the source files, thus being open for all computer platforms. ISO Standard C [6], defined by ISO/IEC 9899:1990, fulfills both requirements. A compiler for Standard C is available on every operating system; C is very flexible with regards to data manipulation and memory handling.

APL Data Structure

The most important data structure is the definition of APL arrays, as shown in figure 3. The three

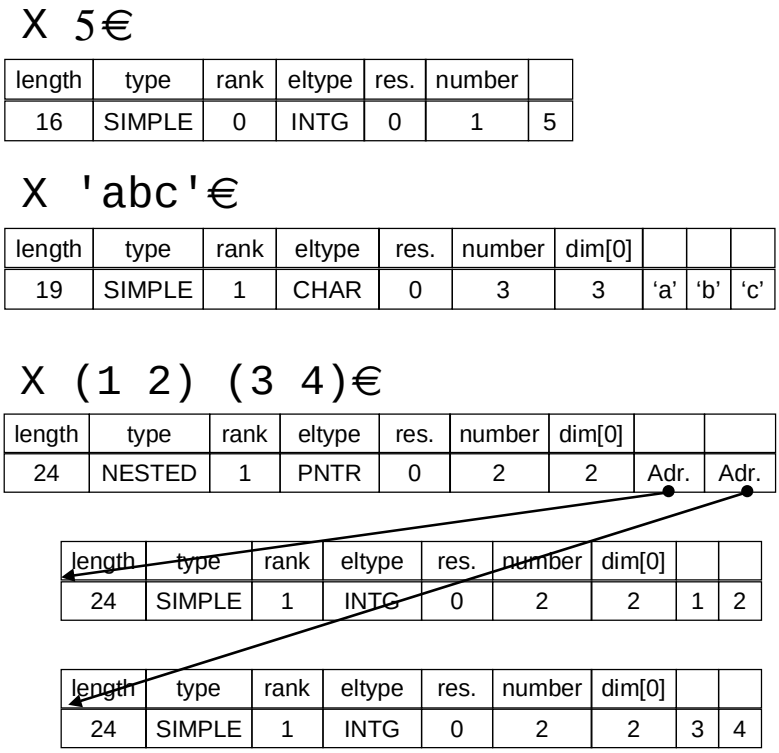

Figure 4: Examples for the representation of two simple and one nested APL array in memory according to the type definition shown in figure 3.

```
typedef struct
{
      long           length;   // total length in bytes
      unsigned char  type;     // SIMPLE or NESTED
      unsigned char  rank;     // Dimension
      unsigned char  eltype;   // INTG, CHAR, REAL, …,PNTR
      unsigned char  reserved; // unused
      long           number;   // total number of elements
      long           dim[1];   // dimension(s), dep. on rank
      //             ...       // Data (dynamic)
} aplarray;
```

Figure 3: The definition of an APL array in C. The field 'dim[]' has, in fact, 'rank' elements or and the array's data directly follow the last 'dim' element.

different examples in figure 4 explain how the APL arrays are represented in the memory. Simple arrays are stored continuously in one memory block. Nested arrays have one memory block for the root array and one for every sub-array. The APL2C system supports the following basic data types: Boolean, character, integer, short integer and real (float).

C Prototypes for APL primitives

All APL primitive functions show identical prototypes. The prototype definition is displayed in figure 5. For niladic functions, all parameters are equal to the null pointer (NULL), and for monadic functions the argument 'larg' is NULL. The proto-type definition for operators is shown in figure 6.

Symbol table

For every APL system a so-called *symbol table* is needed; here the meaning and values for each symbol within the workspace are defined (see figure 7). The symbol table allows one to define local and pseudo local variables. When an APL function is entered for execution, the interpreter saves the actual contents of the symbol table for all local variables. The contents will be restored as soon as the interpreter leaves the executed function. In some APL systems implementations, the symbol table is limited in size (e.g. to 32678 symbols).

Symbol Name	Type	Address
'M'	Variable	● →
'TEST'	Function	● →
'MYOP'	Operator	● →
'DX'	Variable	● →

Figure 7: For every symbol name within the workspace the symbol table keeps the object type and the object's memory address.

However, the symbol table of the APL2C system is only limited by the size of free memory available. To accelerate the execution speed of an APL function, all symbol names were replaced by the corresponding index to the symbol table within the internal representation of the function.

The Compiler

An overview of the APL2C system is given in figure 8. The top level is the APL workspace the basis of which is the APL interpreter system. The latter includes the editor for function definition, the session manager, the handler for the workspace with all its system functions as well as the interpreter for the execution of the APL code.

The idea behind the compiler is to automatically convert an APL function into Standard C code and to replace the interpreter by direct calls to the APL primitives, coded in Standard C, too. This

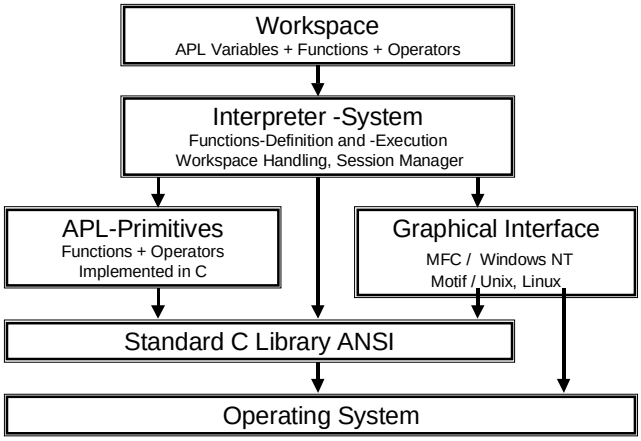

Figure 8: APL2C system structure. The interpreter is based on the Standard C library and on the graphical interface which, however, depends on the operating system.

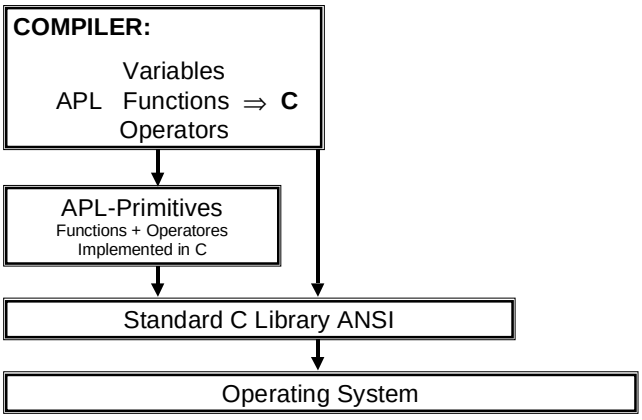

Figure 9: The compiler automatically replaces the interpreter by direct calls to the APL primitives. The resulting code is only based on the Standard C library, but requires the C coded primitive functions and operators (e.g. as library).

calls without axis the corresponding parameters are NULL. ‘elptr’ is a data structure that can hold simple or derived functions (i.e. operator together with operands).

concept is shown in Figure 9. Once the C code has been generated, it can be easily mixed with other C source code files. Figure 10 points out the integration of compiled APL code with other C, C++ sources files.

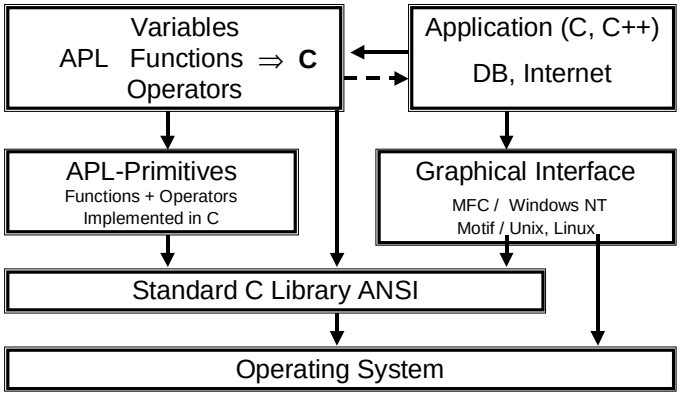

Figure 10: Integration of compiled APL programs into applications written in C, C++.

Syntax Analysis and Symbol Names

Fortunately, APL programs show a simple syntax, i.e. APL statements do not span over different lines. This facilitates the implementation of a syntax analyser representing an important part of the compiler. Below the existing syntactical combinations are shown:

D	data (e.g. variable or literal)
F	niladic function F
F[A]D	Monadic function F (with optional axis)
DF[A]D	Dyadic function F (with optional axis)
F[A]M[A]D	Monadic operator M with monadic function F and optional axes
DF[A]M[A]D	Monadic operator with dyadic function F and optional axes
DF[A]OF[A]D	Dyadic operator with dyadic functions F and optional axes

As a complication, an operator together with its functional operands is forming a so-called *derived function*, which again may serve as functional argument for an operator.

To find out which of the above-mentioned combinations is valid within a line of APL code, the meaning of every referenced symbol must be known. At compile time, the different symbols within an APL instruction pose a problem because, even though the symbol table of the APL2C system may be used, the meaning of those symbols is not well defined and may be ambiguous. A function defined as 'XYZ' can be covered by a local variable named 'XYZ' in one of the functions within the execution stack. To solve this problem, the following restriction is relevant to the functions to be compiled. It is impossible to cover the names of defined APL functions with pseudo-global variables, i.e. local variables defined in one of the functions within the execution stack below the current function. We assume, for example, that a function with name 'TEST' was defined. In this context, it is allowed to define a function as ‘FUNC1’, which has a local variable with the same name 'TEST'. The problem arises when 'FUNC1' is calling a second function 'FUNC2', and 'FUNC2' tries to access the pseudo-global variable 'TEST' of the first function. The compilation of function 'FUNC2' will assume, that

'TEST' refers to function 'TEST' instead of a local variable of another function. The result is a compile time or run-time error.

Restrictions

The following restrictions are applicable to the compilation of any APL function or operator:

The first restriction has already been mentioned above. The names of pseudo-global variables must differ from any other function or operator name within the same workspace.

Since the compiled code has no interpreter, the execution of APL character strings is obviously not possible. For the same reason, it is not allowed to dynamically create new APL functions using the `FX` command.

The last restriction is based on the compiler implementation. If the compiled APL functions refer to global variables within the workspace, the compiler can generate initialization code for these variables. The global variables will be set to their values at compile time but the current version of the compiler only allows simple data for initialization. Nested global arrays cannot be initialized and will cause a compile time error.

Code Generation

The compiler is part of the APL2C system and can be started by the system function `APL2C`. The name of the APL function to be compiled must be given as left argument to the function. The name of a directory for the generated C code files is the right argument.

The compiler will recursively compile all functions that have been used by the function given as left argument. Every APL function or operator will be converted into one C code file. The file name is the name of the APL function to be compiled with an ‘apl_’ prefix. The corresponding C function will take on the same name.

Within the APL2C system the C function name for every APL primitive function is automatically registered at start-up. Thus, it is ensured that for every primitive or system function the corresponding C code function name is known and can be used by the compiler.

Figure 11 shows a small APL program ‘TEST’ computing the square root out of the squares’ sum of the left and right argument. The corresponding C code file ‘apl_TEST.c’ is shown in figure 12 (exactly as automatically created). Line #5 defines the C function apl_TEST. Here, the same prototype definition for APL primitives is used as described under figure 5. Line #11 is the representation (as octal string) of the value 0.5 in the APL array format described in figure 3 and 4. At lines #14 and #15 the actual contents of the symbol table for symbols A and B is stored in the array localvars[] and then the values of A and B are set to the left and right argument. Lines #16 and #17 save the current values of symbols E and L and then clear the contents of these variables. Every APL code line is implemented in its own switch statement to be able to make computed gotos’ where the

```
E←A TEST B;L
L←0.5
E←((A×A)+B×B)*L
```

Figure 11: Small APL program serving as example for compilation into C code. The compiled C code is shown in figure 12.

jump address can dynamically be computed. An APL jump command will modify the C variable ‘linenum’ and will determine the next line of C code to be executed.

```
[  1 ]  #include "includes.h"
[  2 ]  #include "apl2c.h"
[  3 ]  #include "test.h"
[  4 ]
[  5 ]  aplarray *apl_TEST(aplarray *_axis, aplarray *apl_A, aplarray *apl_B)
[  6 ]  {
[  7 ]    aplarray *apl_E;
[  8 ]    symbol localvars[4];
[  9 ]    unsigned char typeof_apl_A, typeof_apl_B;
[ 10 ]   long *tlinenum_ptr, linenum = 0;
[ 11 ]   static char const0[] = "\20\0\0\0\1\0\4\0\1\0\0\0\0\0\0\77";
[ 12 ]
[ 13 ]
[ 14 ]   typeof_apl_A = Initarg(localvars+0, apl_A_idx, apl_A);
[ 15 ]   typeof_apl_B = Initarg(localvars+1, apl_B_idx, apl_B);
[ 16 ]   Initvar(localvars+2, apl_E_idx);
[ 17 ]   Initvar(localvars+3, apl_L_idx);
[ 18 ]   tlinenum_ptr = linenum_ptr;
[ 19 ]   linenum_ptr = &linenum;
[ 20 ]
[ 21 ] nextline:
[ 22 ]   switch(++linenum)
[ 23 ]   {
[ 24 ]     case 1:
[ 25 ]       assign(apl_L_idx, (aplarray *)const0);
[ 26 ]       goto nextline;
[ 27 ]     case 2:
[ 28 ]       assign(
[ 29 ]         apl_E_idx,
[ 30 ]         power(
[ 31 ]           NULL,
[ 32 ]           add(
[ 33 ]             NULL,
[ 34 ]             mul(NULL, VARIABLE(apl_A_idx), VARIABLE(apl_A_idx)),
[ 35 ]             mul(NULL, VARIABLE(apl_B_idx), VARIABLE(apl_B_idx))
[ 36 ]           ),
[ 37 ]           VARIABLE(apl_L_idx)
[ 38 ]         )
[ 39 ]       );
[ 40 ]       goto nextline;
[ 41 ]     default: goto end;
[ 42 ]   }
[ 43 ] end:
[ 44 ]   Resumearg(localvars+0, apl_A_idx, typeof_apl_A, apl_A);
[ 45 ]   Resumearg(localvars+1, apl_B_idx, typeof_apl_B, apl_B);
[ 46 ]   apl_E = return_var(apl_E_idx);
[ 47 ]   Resumevar(localvars+2, apl_E_idx);
[ 48 ]   Resumevar(localvars+3, apl_L_idx);
[ 49 ]   linenum_ptr = tlinenum_ptr;
[ 50 ]   return(apl_E);
[ 51 ] }
```

Figure 12: Resulting C code after compilation of the APL program shown in figure 11. (Explanation, see text above). The C code has been created exactly as shown here including the indenting.

Line #25 assigns the value 0.5 to variable L. Lines #28 to #39 show the C code for the second line of APL code. The C macro VARIABLE(IDX) is reading the memory address of an array out of the symbol table at index IDX. The functions ‘mul’, ‘add’ and ‘power’ are the registered names of the C functions for the corresponding APL primitives. Lines #44 and #45 clear the contents of variables A and B and restore their contents using the localvars[] array. Line #46 is preparing the result and lines #47 and #48 restore the variables E and L to their initial value when the function has been entered.

After all referenced APL functions have been compiled, a makefile will also be created allowing an easy compilation and linkage of the generated code.

Correctness of the Compiler

To verify the correctness of the compiler, it was applied to a complex image-processing task developed with the APL2C system. A total of 106 APL functions with 1,690 lines of APL code (comments included) were compiled into 717 kilobytes (more than 28,000 lines) of C code. For the C code compilation three different compilers (Gnu-C compiler on Linux, Watcom C/C++ Version 10.6 and Microsoft Visual C/C++ Version 5 on Microsoft Windows) were used. Then, the results of the compiled programs were checked; no difference to the results of the interpreted APL program could be made out.

Performance Measurements

It is the concept of this paper’s compiler to replace the interpreter by automatically converting APL code into C code using direct calls to the C coded APL primitive functions. The question is, if and how much faster the compiled C code is compared to the interpreted APL program. Theoretically, the compiled C code should be at least as fast as the interpreter should. The maximum amount of time to be saved is the execution time, used by the interpreter itself. The bigger the data arrays handled by the APL program, the lesser time is needed for the interpretation compared to the data processing by the APL primitive functions.

The two examples below show a comparison between compiled C code and interpreted APL code. The first example is an APL program with a maximum of interpreter load: An empty loop:

```
LOOP;N
N←0
LB:
→(100000>N←N+1)/LB
```

In the second example the following small program was used to determine all prime numbers smaller than 1,000,000 (Eratosthenes’s sieve algorithm):

```
P←PRIM N;⎕IO;J;I;S
⎕IO←1
P←⍳N
S←N*0.5
J←2
LOOP:
I←J×1↓⍳⌊N÷J
P[I]←0
J←J+1+2|J
→(J≤S)/LOOP
P←1↓(P≠0)/P
```

For both examples, the table below displays the resulting execution times measured on a 450 MHz Pentium II computer:

Program	Interpreted	Compiled	Saving
LOOP	3,64 sec	1,90 sec	52 %
PRIM	2,10 sec	2,08 sec	1 %

These measurements confirm, that the compilation saves only the execution time of the interpreter itself.

References

[1] IVERSON K.E.: *A Programming Language*, John Wiley & Sons, Inc., New York (1962)

[2] DISCROLL G.C.JR.; ORTH D.L.: *Compiling APL: The Yorktown APL Translator*, IBM Journal of Research and Development 30(6): 583-593 (1986)

[3] CHING W.-M.: *Program Analysis and Code Generation in an APL/370 Compiler*, IBM Journal of Research and Development 30(6): 594-602 (1986)

[4] BUDD T.: *An APL compiler*, Springer-Verlag New York Berlin Heidelberg London Paris Tokyo, ISBN: 0-387-96643-9 (1988)

[5] BERNECKY R.: *An Overview of the APEX Compiler*, University of Toronto, Department of Computer Science, Technical Report 305/97

[6] ISO/IEC 9899:1990: *ISO C Standard*, ISO Central Secretariat, Case postale 56, 1211 Geneva 20, SWITZERLAND