ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-185

A Collection of Tools in Support of Automatic Differentiation

by

Andrew Mauer

Mathematics and Computer Science Division

Technical Memorandum No. 185

February 1994

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

A Collection of Tools in Support of Automatic Differentiation

Andrew Mauer

Abstract

This document contains a collection of notes about tools that we have found useful in our work on automatic differentiation.

- *Using m4 for Procedure Renaming.* Most transformations necessary to link C and Fortran programs involve changing the case of the C prodecure names and some other trivial manipulations. We automate this procedure.
- *On Linking ADOL-C and Fortran Programs.* This portion of the document serves a dual purpose. It is a guide to getting started with ADOL-C, and it also describes methods of linking ADOL-C and Fortran programs together.
- *A Quick Discussion of the* `fortran-manipulate.pl` *Package.* We provide two low-level `perl` functions that aid in coping with the fact that Fortran "logical" lines may include an initial line and many continuation lines.
- *Fortran Text Manipulation with* `perl`. We describe a very powerful `perl` template that may be easily customized to perform many common Fortran manipulations, such as expansion of various templates in the code.
- *A Simple Wrapper for ADIFOR.* Some simple transformations of the ADIFOR script and composition files allow much more intuitive syntax.

1 Using m4 for Procedure Renaming

It is possible to use the standard Unix program `m4` to rename C procedures in an intelligent and portable manner that also has the major benefit of being simple to maintain. In our experience, the only transformations necessary to have a C subroutine callable from Fortran are the conversion to upper or lower case and the prepending or appending of one or more underscores. Changing case is beyond the scope of the C preprocessor but is possible in `m4`, hence the choice.

Many sections of this document are written in the literate programming system `noweb` [2] based on Donald Knuth's original WEB. It is designed to blend the program code with an intelligible explanation of what occurs. The program chunks presented here are tagged with a number and a letter; the number corresponds to the page number on which the chunk is found, and the letter corresponds to the chunk's position on that page. After each chunk, there appears a listing of which chunks the current one is used in and, if the chunk is defined in more than one place, where the rest of the definitions are.

1.1 User Interface

The basic idea is that we will have an `m4` program that transforms to upper or lower case the C identifiers by generating C preprocessor `#define` macros. Our base C code will read normally, regardless of the renaming necessary.

```
#include "rename.h"

void
c_in_fortran (double *argument)
{ ... }
```

The include file will be generated automatically by `m4` from a template file. The template file will list all of the identifiers that we wish to rename, each wrapped in a call to the `m4` function `fort`. (This include file may be automatically generated if desired.) It will resemble the following segment.

"template-file" 2 ≡

```
#ifndef FORTRAN_TO_C_TEMPLATE_H
#define FORTRAN_TO_C_TEMPLATE_H
fort(c_in_fortran)
fort(another_c_subroutine)
...
#endif /* ndef FORTRAN_TO_C_TEMPLATE_H */
```

◇

Note: The `#ifndef` bracketing is strongly suggested because it prevents the resulting C header file from being included more than once in a given program.

Note: The user should be careful about adding any other information to the template file, as `m4` may interpret it rather than simply passing it through as expected.

To generate the C header file, the user will invoke `m4` *with the proper architecture defined.* Currently, we support `sun4` (Sun 4 and SPARCstation, SunOS 4.1.3), `rs6000` (IBM RS/6000, AIX 3.2, `iris4d` (SGI Indigo, Irix 4.0.5), and `cray` (Cray Y-MP, Unicos 6). Adding others is simple and is covered in the Implementation section below. We give a sample invocation.

```
m4 -Drs6000 renaming-tool template-file > renaming.h
```

Note: Two `m4` processors ship with SunOS 4.1.x. The executable `/usr/5bin/m4` should be used; the other is broken. (Note that `/usr/5bin` is frequently not on one's search path, so the name must be typed in full.) There is a GNU m4 implementation available on most platforms if your `m4` is broken.

After the header file has been generated in this way, its contents will be C preprocessor macros that redefine the basic names given in the template file. A sample renaming file for a Cray is given below.

```
#define c_in_fortran C_IN_FORTRAN
#define another_c_subroutine ANOTHER_C_SUBROUTINE
```

Since the C preprocessor `#define` will affect only whole tokens, and does not affect anything inside of strings, there is little danger of the C code being corrupted by these redefinitions. For instance, a variable named `c_in_fortran_var` will *not* be renamed by the above code.

1.2 Implementation

This section describes the `m4` macros that are used to provide the functionality described above.

1.2.1 User-Serviceable Macros

The macro `fort` generates a C preprocessor define statement that renames its first argument properly. Note that the `#define` portion is quoted twice, so that on reevaluation the `#` character will not make `m4` think that it is reading a comment.

⟨`fort` macro 3a⟩ ≡

```
define(‘fort’,‘‘#define ’$1 fortran_identifier($1)’’)
◇
```

Macro referenced in scrap 4f.

The `fortran_identifier` macro is the trick to success. We define it differently depending on the architecture desired. Note that the second argument to the ifdef must be *quoted* or it will not be evaluated in the expected manner. In `m4`, quoting opens with the left single quote ‘‘, and closes with the right single quote ’’, so it nests without a problem.

We will give a synopsis of all the computers supported and then briefly show each.

⟨All supported architecture definitions 3b⟩ ≡

```
⟨Warning if no architecture is defined 3e⟩
⟨Sun definition of fortran_identifier 3a⟩
⟨rs6000 definition of fortran_identifier 3b⟩
⟨cray definition of fortran_identifier 3d⟩
⟨irix4d definition of fortran_identifier 3c⟩
◇
```

Macro referenced in scrap 5c.

- sun4 (SunOS 4.1.x): We wish lowercase identifiers with an appended underscore.

  ⟨Sun definition of `fortran_identifier` 3c⟩ ≡

  ```
  ifdef(‘sun4’,
          ‘define(‘fortran_identifier’,
                  ‘m4_append_underscore(m4_lowercase($1))’)’)
  ◇
  ```

  Macro referenced in scrap 3b.

- rs6000 (AIX): We wish lowercase identifiers with no appended underscore.

⟨rs6000 definition of `fortran_identifier` 4a⟩ ≡

```
ifdef(‘rs6000’,
        ‘define(‘fortran_identifier’,
                ‘m4_lowercase($1)’)’)
```

◇

Macro referenced in scrap 3b.

- irix4d (Indigo Irix 4.0.5): We wish lowercase identifiers with an appended underscore.

⟨irix4d definition of `fortran_identifier` 4b⟩ ≡

```
ifdef(‘iris4d’,
        ‘define(‘fortran_identifier’,
                ‘m4_append_underscore(m4_lowercase($1))’)’)
```

◇

Macro referenced in scrap 3b.

- cray (Cray Y-MP Unicos 6): We wish uppercase identifiers with no appended underscore.

⟨cray definition of `fortran_identifier` 4c⟩ ≡

```
ifdef(‘cray’,
        ‘define(‘fortran_identifier’,
                ‘m4_uppercase($1)’)’)
```

◇

Macro referenced in scrap 3b.

- No architecture defined. If the user does not define an architecture, we issue a warning message.

⟨Warning if no architecture is defined 4d⟩ ≡

```
define(‘fortran_identifier’,
       ‘Need to define some architecture with -D$(ARCH)’)
```

◇

Macro referenced in scrap 3b.

1.2.2 Under-the-Hood Macros

We need support functions to put words into upper or lower case. This task is done through a `tr`-like mechanism provided by `m4`.

⟨Support Functions 4e⟩ ≡

⟨m4 Uppercase Function 4f⟩
⟨m4 Lowercase Function 5a⟩
⟨m4 Underscore Appending Function 5b⟩
◇

Macro referenced in scrap 5c.

This function translates all lowercase letters in its first argument to uppercase letters.

⟨m4 Uppercase Function 4f⟩ ≡

```
define(‘m4_uppercase’,‘translit($1,abcdefghijklmnopqrstuvwxyz,ABCDEFGHIJKLMNOPQRSTUVWXYZ)’)
```

◇

Macro referenced in scrap 4e.

This function translates all uppercase letters in its first argument to lowercase letters.

⟨m4 Lowercase Function 5a⟩ ≡

```
define(‘m4_lowercase’,‘translit($1,ABCDEFGHIJKLMNOPQRSTUVWXYZ,abcdefghijklmnopqrstuvwxyz)’)
◇
```

Macro referenced in scrap 4e.

This function appends an underscore to its first argument.

⟨m4 Underscore Appending Function 5b⟩ ≡

```
define(‘m4_append_underscore’,‘$1_’)
◇
```

Macro referenced in scrap 4e.

Of course, the whole package needs to be put together.

"renaming-tool" 5c ≡

```
⟨fort macro 3a⟩
⟨All supported architecture definitions 3b⟩
⟨Support Functions 4e⟩
◇
```

2 On Linking ADOL-C and Fortran Programs

It is important that you skim the ADOL-C manual [1]. It has most of the basics. I elaborate here only the representation of the power series.

The gradient objects corresponding to f is the n-th derivative of f with respect to x divided by n factorial. That is, they are the coefficients on the x terms in the Taylor expansion of f.

$$a_0x^0 + a_1x^1 + (a_2/2!)x^2 + (a_3/3!)x^3 \cdots$$

2.1 An Easy ADOL-C Program

Let us examine a simple ADOL-C program to get the derivative of the function $f(x) = x^2$ with respect to x. The overall structure of our program will resemble the following.

"adol-c-example-1.cc" 6a ≡

```
⟨Obligatory Includes 6b⟩
⟨Incidental Includes 6c⟩
⟨Function Definition 6d⟩
⟨Derivative Subprogram 7a⟩
⟨Main Program 10b⟩
◇
```

Only two files are necessary to include in order to use ADOL-C.

⟨Obligatory Includes 6b⟩ ≡

```
#include "adouble.h"
#include "adutils.h"
◇
```

Macro referenced in scrap 6a.

We include one more header file so we can do I/O.

⟨Incidental Includes 6c⟩ ≡

```
#include <iostream.h>
◇
```

Macro referenced in scrap 6a.

The actual definition of the function is similarly trivial; one needs only to take care to use the type `adouble` in the place of `double`. To my knowledge, single-precision (`float`) arithmetic is not available. (The automatic promotion of a `float` to a `double` by the compiler makes this a non-issue.) Note that the function returns an `adouble`, not a `double`.

⟨Function Definition 6d⟩ ≡

```
adouble
function (adouble x)
{
     adouble result;
     result = x * x;
     return result;
}
◇
```

Macro referenced in scrap 6a.

The difficult part is correctly setting up the call so that it will compute the desired derivatives. Since our function takes one input, we will pass this one input from the main program to our derivative-computing subroutine.

⟨Derivative Subprogram 7a⟩ ≡

```
void derivative(double x, double &result)
{
  ⟨Variable Initializations 7b⟩
  ⟨Set up "tracing" 9b⟩
  ⟨Call function 9c⟩
  ⟨End "tracing" 9d⟩
  ⟨Compute Derivatives 9e⟩
  ⟨Print results 10a⟩
}
```

◇

Macro referenced in scrap 6a.

2.1.1 Variable Initializations

There are a number of important variables that we will initialize. For the most part, they need not be initialized independently, but all must appear in the function call that computes the desired derivatives. The name per se is not significant; the ones used here are the ones used in the manual.

⟨Variable Initializations 7b⟩ ≡

```
⟨Tape Tag 7c⟩
⟨Number of Independents and Dependents 7d⟩
⟨Highest Order of Derivatives Desired 7e⟩
⟨Taylor Coefficients of Intermediate Quantities to Keep 8a⟩
⟨Declare Dependent and Independent Gradient Object 8b⟩
⟨Allocate Dependent and Independent Gradient Object 8c⟩
⟨Initialize Dependent and Independent Gradient Object 8d, ... ⟩
```

◇

Macro referenced in scrap 7a.

The variable `tag` is an integer indicator of which "stream" one is working with. It is possible to write and process more than one tape, so one may keep two or more sets of data from which to compute derivatives. For casual use, it should just be set to zero.

⟨Tape Tag 7c⟩ ≡

```
int tag = 0;
```

◇

Macro referenced in scrap 7b.

One must specify the number of dependent and independent variables. These can be compile-time constants or set at run time, but the latter requires dynamic allocation of the gradient objects. We will do the latter.

⟨Number of Independents and Dependents 7d⟩ ≡

```
int number_of_independents = 1;
int number_of_dependents = 1;
```

◇

Macro referenced in scrap 7b.

One also needs to specify the highest order of derivatives desired.

⟨Highest Order of Derivatives Desired 7e⟩ ≡

```
int derivative_order = 2;
```

◇

Macro referenced in scrap 7b.

We only need to keep intermediate Taylor coefficients if we are interested in the reverse mode.

⟨Taylor Coefficients of Intermediate Quantities to Keep 8a⟩ ≡

```
int keep = 0;
```
◇

Macro referenced in scrap 7b.

One needs space for the gradient objects. They are either declared as two-dimensional arrays or created with the C++ `new` operator. Note that these are `double` variables, not `adouble`.

⟨Declare Dependent and Independent Gradient Object 8b⟩ ≡

```
double **X_indeps;  // Independent
double **Y_deps;  // Dependent
```
◇

Macro referenced in scrap 7b.

We will immediately allocate memory for the gradients. Note the allocation of `derivative_order+1` elements; we must allow for the "zero order derivative."

⟨Allocate Dependent and Independent Gradient Object 8c⟩ ≡

```
int loop;

X_indeps = new double* [number_of_independents];
for (loop = 0 ; loop < number_of_independents; loop ++)
    X_indeps[loop] = new double[derivative_order+1];

Y_deps = new double* [number_of_dependents];
for (loop = 0 ; loop < number_of_dependents; loop ++)
    Y_deps[loop] = new double[derivative_order+1];
```
◇

Macro referenced in scrap 7b.

Of course, we wish to initialize the independents.[1] To have an independent variable behave as expected, one sets the first partial derivative to 1. No initialization of the dependents is necessary.

⟨Initialize Dependent and Independent Gradient Object 8d⟩ ≡

```
for ( int j = 0 ; j < number_of_independents ; j ++ )
{
     for (int i = 0; i < derivative_order ; i++)
     {
          X_indeps[j][i] = 0.0;
          if ( i == 1 )
          {
               X_indeps[j][i] = 1.0;
          }
     }
}
```
◇

Macro defined by scraps 8d–9a.
Macro referenced in scrap 7b.

When initializing the independents, one should set the zero-order partial derivative to the value of the independent variable.

[1] One can think about power series and get effects similar to the seed-matrix initialization in ADIFOR (not compression; only multiplication by a constant).

⟨Initialize Dependent and Independent Gradient Object 9a⟩ ≡

```
// Here we know there is only one independent
X_indeps[0][0] = x;
```
◇

Macro defined by scraps 8d–9a.
Macro referenced in scrap 7b.

2.1.2 Function Computation

The rest of the process is quite simple. We call the "tracing" function to tell ADOL-C to write its tape, call the function, and end the tracing.

⟨Set up "tracing" 9b⟩ ≡

```
trace_on (tag, keep);
```
◇

Macro referenced in scrap 7a.

All of the "active" variables passed into the function must be of type `adouble`. An "active" variable is initialized from its regular `double` counterpart by using the operator `<<=` as below. To extract the `double` value from an active variable, one uses the `>>=` operator.

⟨Call function 9c⟩ ≡

```
adouble active_x;
adouble active_result;

// Initialize the active variable
active_x <<= x;

active_result = function (active_x);

// Extract the result from the active var
active_result >>= result;
```
◇

Macro referenced in scrap 7a.

Once all of the functional computation is completed, `trace_off` should be called.

⟨End "tracing" 9d⟩ ≡

```
trace_off();
```
◇

Macro referenced in scrap 7a.

2.1.3 Getting Derivative Values

To get derivative values, the user should call one of the functions detailed in the manual. For our purpose, `forward` is sufficient.

⟨Compute Derivatives 9e⟩ ≡

```
forward(tag, number_of_dependents, number_of_independents,
        derivative_order, keep,
        X_indeps, Y_deps);
```
◇

Macro referenced in scrap 7a.

The array `Y_deps` contains the Taylor series of the dependent variables in chronological order of their designation with the `>>=` operator (starting with `Y_deps[0]`). The entry `Y_deps[n][0]` is the value of the dependent number `n`. The entry `Y_deps[n][i]` is the i-th derivative of dependent `n` with respect to the independent.

There is no allowance for multivariate functions in the old (Version 1.3) forward mode.[2] In a simple experiment, I analyzed the function $f(x, y) = x * y$, for $x = 1.5$, $y = 6.0$, with x having the Taylor series 1.5, 1, 0, and y having the Taylor series 6.0, 1.0, 0.0. The resulting derivative of f had the Taylor series 9.0, 7.5, 1.0. Version 1.4, however, has a new function, `hov_forward`, that computes the "higher-order vector" in forward mode. I am told that this is capable of handling independents.

2.1.4 Printing Results

We will use a simple output routine to print out the Taylor coefficients of each of the dependents.

⟨Print results 10a⟩ ≡

```
for (int k = 0; k < number_of_dependents ; k++ )
{
     cout << "Dependent #" << k << " has stored Taylor series coefficients: " << endl;
     for (int l = 0 ; l < derivative_order+1; l++)
     {
          cout << Y_deps[k][l];
          if ( l != derivative_order )
               cout << ", ";
     }
     cout << endl;
}
```

◇

Macro referenced in scrap 7a.

2.1.5 Main Program

A trivial main program is needed to drive our subroutine. We will call it with an arbitrary value for the function.

⟨Main Program 10b⟩ ≡

```
int main (void)
{
     double result;

     derivative(7.0, result);

     // Safe exit
     return 0;
}
```

◇

Macro referenced in scrap 6a.

2.2 Connecting ADOL-C and Fortran

ADOL-C can be used with Fortran in two ways. One way is to set up a C++ process that acts as a server, getting input from the Fortran program and returning the output of some function along with the relevant derivatives. This was the first approach that we used. Another way is to link together a Fortran program and a C++ program; this is the method to which we will devote the most attention. It has been tested on a Sun SPARCstation, under Sun OS 4.1.3, and on an RS/6000 workstation under AIX, both with the

[2] It seems that all of the Taylor series are written in terms of the same variable x. When one gives the Taylor series for a given independent, ADOL-C assumes the user is telling it how that variable varies with respect to some underlying independent.

GNU `g++` compiler and the native Fortran 77 compiler. A similar method should work with other operating systems..

> *Warning*: No C++ global constructors or destructors will be called if the main routine in the resulting program is from Fortran. The Fortran I/O libraries will not be initialized if the resulting main routine is from C++.

2.2.1 Joining by Named Pipes

Basically, the method of joining by named pipes involves setting up two processes, a server and a client, that communicate by named pipes. Generally the C++ side will be the server, accepting input values for functions and returning the function output and the desired derivatives.

The way we set up our communications was quite simple. Two named pipes were created, `inpipe` and `outpipe`.

```
mknod inpipe p
mknod outpipe p
```

The C++ server was set up so that it read from the standard input and wrote to the standard output. It was then invoked in the background reading from `inpipe` and writing to `outpipe`.

```
./c++-derivative-server < inpipe > outpipe &
```

On the (Fortran) client side, one writes function input to `inpipe` and reads results from `outpipe`. We implemented this by linking a C routine with Fortran and using the standard C functions for I/O. Presumably the Fortran I/O functions would serve the same purpose.

An example is available in `/home/derivs/share/fortran-to-c/pipe-communication`.

2.2.2 Joining by Linking Together

There are two basic steps one must follow to link a C++ routine and a Fortran routine. First, one needs to create a subroutine on the C++ side that is accessible through Fortran. Since any C++ identifier is "mangled" in the output (the mangling process is a compiler-dependent change of the name to allow overloading), the routine should be declared `extern "C"`. In addition, it may need to have a specific case, and it will probably need an underscore appended (on the C++ side only). See Section 1 for further information.

- Cray: Uppercase, no appended underscore.
- rs6000: Lowercase, no appended underscore.
- sun4: Lowercase, appended underscore.
- iris4d: Lowercase, appended underscore.

For instance, on the Fortran side, we will have

```
call deriv (x,y)
```

whereas on the C++ side for a SUN, we will have

```
extern "C" deriv_ (double *x, double *y)
{ /* Function Body */ }
```

and on the Cray, we will have

```
extern "C" DERIV (double *x, double *y)
{ /* Function Body */ }
```

The individual program modules are then compiled separately.

The second major step in the process is to link them together. This must be done with the C++ linker since nontrivial things happen to a C++ program in the link stage. In general, linking will involve adding the Fortran libraries to the C++ link step. Here we detail how to find those magical libraries. If one is using an RS/6000 or Sun, the remainder of this (sub)section can be skipped.

First, we create or get a trivial Fortran program called `test.f`. The following will serve nicely.

```
program main
end
```

Now we compile this program with the appropriate Fortran compiler (usually `f77` or `xlf`), but using the `-v` switch so that we see everything that happens.

IBM RS/6000

On an rs6000, a verbose link shows

```
% xlf -v test.f
exec:
/usr/lpp/xlf/bin/xlfentry(xlfentry,test.f,/tmp/F8HAID0V,test.lst,\
xlfsmsg.cat,xlfmsg.cat,NULL)
** main   === End of Compilation 1 ===
1501-510  Compilation successful for file test.f.
exec: /bin/ld(ld,-bh:4,-T512,-H512,/lib/crt0.o,test.o,-lxlf,-lm,-lc,NULL)
unlink: test.o
```

The important lines are those in the link step (`/bin/ld`). Scanning through the arguments, we see that (1) no libraries are being added to the search path with the `-L` option; and (2) the options that are linking libraries are `-lxlf`, `-lm`, and `-lc`. We will add these three libraries to the C++ link step.

Note: We do not add `crt0.o`.

Sun SPARCstation

On a Sun 4, a verbose link shows

```
% f77 -v test.f
/usr/lang/SC1.0.1/f77pass1 "-P -cg87" test.f /tmp/f77pass1.17110.s.0.s \
/tmp/f77pass1.17110.i.1.s /tmp/f77pass1.17110.d.2.s
test.f:
 MAIN main:
/usr/lang/SC1.0.1/as -o test.o -Q -cg87 /tmp/f77pass1.17110.s.0.s \
/tmp/f77pass1.17110.i.1.s /tmp/f77pass1.17110.d.2.s
rm /tmp/f77pass1.17110.s.0.s
rm /tmp/f77pass1.17110.i.1.s
rm /tmp/f77pass1.17110.d.2.s
/bin/ld -dc -dp -e start -u _MAIN_ -X -o a.out /usr/lang/SC1.0.1/crt0.o \
/usr/lang/SC1.0.1/cg87/_crt1.o -L/usr/lang/SC1.0.1/cg87 \
-L/usr/lang/SC1.0.1 test.o -lF77 -lm -lc
rm test.o
```

Again, the important data here is (1) two directories have been added to the library search path: `-L/usr/lang/SC1.0.1/cg87` and `-L/usr/lang/SC1.0.1`; and (2) three libraries are being linked: `-lF77`, `-lm`, and `-lc`. We will add both the libraries and the library search path to the C++ link step.

Note: We do not add `crt0.o`.

2.2.3 Specific (Practical) Results

First, a summary of the known ways to link Fortran and C++. To link on an RS/6000, one should use a command of the form

```
[Usual link line plus...] -lxlf -lm
```

To link on a sun, one should use a command of the form

```
[Usual link line plus...] \
-L/usr/lang/SC1.0.1/cg87 -L/usr/lang/SC1.0.1 -lF77 -lm
```

The user should re-read Section 2.2.2, page 11, to become familiar with the Fortran-C++ calling sequence. All Fortran double-precision variables are passed as `double *` to C++. Strings in Fortran may or may not present difficulties, depending on the compiler.[3] Remember that we have the normalized Taylor coefficients, not the plain derivatives (one must to multiply them by the order factorial to get plain derivatives).

[3] We encountered no problems receiving them as `char` in C++ on the RS/6000, Sun 4, and SGI Irix Indigo machines but did have difficulties on the NeXT.

3 A Quick Discussion of the `fortran-manipulate.pl` Package

The `fortran-manipulate.pl` package provides two basic functions: "unify" and "flow". The function "unify" assembles a single logical line from an initial line and continuation lines. The function "flow" breaks a single logical line into continuation lines suitable for digestion by a Fortran compiler.

3.1 `unify`

CALLING: `&unify (*array_of_fortran_lines)`

INPUT: A single array of 1 line, its continuation lines, and optionally more lines that are ignored. The initial line should be in the first index of the array (that is, in `$[`).

RETURNS: The first line with all of its continuation lines, ending with a newline.

Note: The `*name` calling format is necessary to allow the routine to modify the array that is passed to it.

This routine strips out the continuation line "junk" (the spaces and continuation line character — up to column 6) and anything after column 72 and then appends the multiple lines in the array together into one long line. This line is returned.

The lines are removed from the argument array as they are added to the final assembly. Hence the array passed in *will be modified.* (This behavior can be changed, if desired.)

This routine also pads short lines to column 72.

3.2 `flow`

CALLING: `&flow($unflowed_line, @nobreak_list)`

INPUT: `$unflowed_line`, one long Fortran line, like that generated by unify(). *It should not have embedded newlines.* `@nobreak_list` is a (possibly empty) array of strings that should be placed on lines by themselves.

RETURNS: `$unflowed_line` broken up into continuation lines. Each one of these lines will end with a newline (including the last one).

Warning: If the string specified has embedded newlines, they will be respected as line breaks. This may be desired in the middle of the line, but there should *certainly not* be one at the beginning. Beginning with a newline will cause the first line to be generated as a continuation. This will most likely result in incorrect code.

The routine "flow" breaks a long line into smaller lines of 72 characters or less, adding the continuation line prefix to every line after the first. Elements of the array `@nobreak_list` are placed on lines by themselves (followed by as many commas, spaces, and left parentheses immediately after it; this behavior can be changed if desired). This is useful for preserving preprocessor tokens.

The arguments are not modified.

3.3 Warning

If one processes a program that uses the C preprocessor to `#define` certain tokens, care must be taken that they are not broken across a continuation line. In practice, this is hardly ever a problem.

4 Fortran Text Manipulation with `perl`

We provide a template and support functions to make it simple to perform certain types of massaging Fortran source code. We process only lines that are in a user-specified format, so we do not need to understand the entire source.

4.1 User-Level Details

The general philosophy of this template is to alter only those lines that the user has specified as "interesting." Other lines are left intact, allowing for any absurd structure such as significant information in columns beyond 72, line numbers on continuation lines, comment code. By passing through the majority of the source code unaltered, we have the greatest chance of not changing (destroying) the original program.

This template uses the `fortran-manipulate.pl` package. Section 3 documents that package.

There is a separate document explaining the functions `unify` and `flow` contained in that package. No knowledge of them is necessary to use this template.

> *Note*: This entire program runs with `$[ = 1`, so arrays are indexed starting at one, just like columns in Fortran lines.

4.1.1 Command Line

By default, the template recognizes four options. The `-help` option prints a help message. The `-version` option prints the RCS version of the template expander (or the derived program, if it is under RCS control). The `-c` option prints all output to the standard output, rather than the specified file (see Section 4.1.2). The `-n` option does not write any output. This can be useful if one prints specific debugging messages but does not care about the actual output.

4.1.2 Output File Names

The name of the output file is derived from the name of the input file in the program section `CHOOSING THE OUTPUT FILENAME`. By default, input files ending in `.f` are mapped to the same base name, with the suffix `.F`. Other files have `.processed.F` appended.

> *Warning*: If the output file name is the same as the input file name, the input file will get clobbered.

By altering the replacement expression or adding new ones, the user can control the name of the output file. Another option that should be considered is that of invoking the tool once for each individual file, using the `-c` option, and redirecting the resulting output. The following example illustrates this style.

```
./template -c myfile.f > processed.F
```

4.1.3 Doing Something Useful

The areas mentioned above will probably be tweaked when customizing the template, but the most work will go into actually making it do something. The first thing one will need to do is change `$interesting_regex`. This is a regular expression that must match the *initial line* of any whole line (initial line and continuation lines) that one desires to "process." It is very important to note that this *must* match the initial line or incorrect results may be obtained.

Once a line matches the `$interesting_regex`, all of its continuation lines are gathered together and joined into one long line, and it is passed to the subroutine `your_routine`.[4] In this subroutine one can restructure the line in any way desired (or even replace it). The line that is returned from this subroutine is taken to be the "processed" line. This processed line is broken into continuation lines and output.

If one wishes to make modifications that will not be subject to the process of folding into continuation lines, for instance adding `#ifdef` directives to conditionally compile code, one should do them in the `your_post_processing` routine. The line that this routine returns is output exactly as it is received.

[4] For those who need to know exactly what happens, columns 1 to 72 of the initial line are taken, and the columns 7 to 72 of all of its continuation lines are appended to it.

4.2 Technical Details of the Main Loop (`process_fortran_file`)

If a line is "interesting," it is pushed on to the `@raw_lines` array, and we see whether it has any continuation lines. We set the `$interesting` flag to indicate that we are not just supposed to pass through continuation lines and go back through the processing loop.[5] We go on, grabbing continuation lines and spewing out any embedded comments, until we find a line that is not a continuation line.

At this point, we "unify" the lines and send the unified line to the routine that processes the exception handler call. We "flow" the line we get back, and output it.

When we near the end of loop after processing an "interesting" line, we are carrying another line in `$_` that may or may not be interesting. We reset `$interesting` to false and go back through the loop to check `$_`.

[5] Of course, if no more lines are in the file, we drop on down to process this line. This is where it is important that `$_` be reset, so that the loop does not continue to think it is getting new interesting material just because it is not reading in anything new.

5 A Simple Wrapper for ADIFOR

We provide some simple massaging of the script and composition files required by ADIFOR Version 1. This makes their use more intuitive and less prone to syntatic errors.

The following actions are performed for both the composition and script files.

- The # character is considered a comment character. Everything from it to the end of the line is stripped out. No provision is made for escaping it.
- Leading and trailing whitespace is stripped from lines, so there is no danger of ADIFOR thinking that your file has several leading or trailing spaces in the name.
- Blank lines are removed, preventing ADIFOR from interpreting them in any way.

The following actions are performed only in the script file.

- Whitespace around commas is removed. This prevents bad things from happening in the input and output variables lists.
- Script file variables may be set with the = operator. This allows the very natural syntax `PMAX=30`.
- The variable `DVARS` is accepted as a synonym for `OVARS`.

We gain some incidental bonuses from using this wrapper.

- The directory `/usr/local/adifor/bin` is added to the search path, so ADIFOR can find its support programs even if they are not on the user's search path.

Other potential enhancements could include allowing "\" at the end of a line to designate the next line as a continuation line.

A Simple ADOL-C Code

```
#include "adouble.h"
#include "adutils.h"

#include <iostream.h>

adouble
function (adouble x)
{
     adouble result;
     result = x * x;
     return result;
}

void derivative(double x, double &result)
{
  int tag = 0;

  int number_of_independents = 1;
  int number_of_dependents = 1;

  int derivative_order = 2;

  int keep = 0;

  double **X_indeps;  // Independent
  double **Y_deps;  // Dependent

  int loop;

  X_indeps = new double* [number_of_independents];
  for (loop = 0 ; loop < number_of_independents; loop ++)
      X_indeps[loop] = new double[derivative_order+1];

  Y_deps = new double* [number_of_dependents];
  for (loop = 0 ; loop < number_of_dependents; loop ++)
      Y_deps[loop] = new double[derivative_order+1];


  for ( int j = 0 ; j < number_of_independents ; j ++ )
  {
       for (int i = 0; i < derivative_order ; i++)
       {
            X_indeps[j][i] = 0.0;
            if ( i == 1 )
            {
                 X_indeps[j][i] = 1.0;
            }
       }
  }
```

```
  // Here we know there is only one independent
  X_indeps[0][0] = x;



  trace_on (tag, keep);

  adouble active_x;
  adouble active_result;

  // Initialize the active variable
  active_x <<= x;

  active_result = function (active_x);

  // Extract the result from the active var
  active_result >>= result;

  trace_off();

  forward(tag, number_of_dependents, number_of_independents,
          derivative_order, keep,
          X_indeps, Y_deps);

  for (int k = 0; k < number_of_dependents ; k++ )
  {
       cout << "Dependent #" << k << " has stored Taylor series coefficients: " << endl;
       for (int l = 0 ; l < derivative_order+1; l++)
       {
            cout << Y_deps[k][l];
            if ( l != derivative_order )
                 cout << ", ";
       }
       cout << endl;
  }

}

int main (void)
{
     double result;

     derivative(7.0, result);

     // Safe exit
     return 0;
}
```

References

[1] Andreas Griewank, David Juedes, and Jay Srinivasan. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. Technical Report MCS-P180-1190, Argonne National Laboratory, 1990.

[2] Norman Ramsey. Literate-programming tools can be simple and extensible. Report at `bellcore.com` in `/pub/norman/noweb/xdoc/ieee.tex`. software at `bellcore.com` in `/pub/norman/noweb-2.5a.shar.z`., Department of Computer Science, Princeton University, October 1993.