ADT - Automatic Differentiation via TAPENADE Utility : Users Manual

May 25, 2018

ADT is a command line application that allows you to produce Pascal units or C++ classes containing automatically differentiated code through the use of the TAPENADE automatic differentiation tool.

Contents

Chapter 1

Building ADT

1.1 Prerequisites

Before building ADT and the demonstration R library you need to ensure that you have TAPENADE installed on your machine. TAPENADE is available from:

```
http://www-sop.inria.fr/tropics/tapenade/downloading.html
```

You can read more about TAPENADE and automatic differentiation here:

```
http://www-sop.inria.fr/tropics/tapenade.html
```

The TAPENADE bin folder needs to be in the PATH environment variable so that it can be run without specifying a fully qualified path to the file. On Linux systems a good place to install TAPENADE is in */usr/share* (eg. */usr/share/tapenade3.10*). On Windows systems install under the Program Files folder (eg. *C:\Program Files\tapenade3.10*).

To build ADT, on Linux distributions you must have gcc, flex, bison and the automake tool set installed and on Windows you must have either Visual Studio 2005 (or higher installed) or cygwin with gcc and the automake toolset installed and flex and bison. Flex and Bison can be found here:

```
http://www.gnu.org/software/flex
```

and

```
http://www.gnu.org/software/bison
```

When installing Flex and Bison on Windows ensure that the path to the install location does not contain spaces as Bison doesn't handle paths with spaces correctly on the Windows build. The default install location is typically *C:\Program Files\GnuWin32*. Change it to *C:\GnuWin32*. The bin folder will also need to be in the system path so add *C:\GnuWin32* to the system path using the Advanced System Settings in the control panel system section.

In order to run the example project you will need to have R installed. R can be obtained from:

```
http://www.r-project.org
```

Further to these requirements, should you wish to build and run the Pascal example then you will also need to install lazarus and ensure that the Free Pascal compiler bin folder is in the PATH environment variable. Lazarus is available from here:

```
http://www.lazarus.freepascal.org
```

1.2 Building ADT on Linux

Run the configure script to generated the necessary makefiles. That is, in the root folder for adt (the folder where the source package is installed - for example: */usr/src/adt-1.03*), run the following in a command shell:

```
./configure
```

On completion run:

```
make clean
```

and,

```
make
```

and,

```
sudo make install
```

If you have compile errors there are a number of possible reasons. Firstly, do you have the necessary tools installed (gcc, flex, bison, TAPENADE and R)? If you do, is TAPENADE and R in the search PATH and is R installed in a different location to where the configure script expects it to be? By default, configure assumes the R include files to be located in,

```
/usr/share/R/include
```

and the R shared library to be located in,

```
/usr/lib/R/lib
```

If your locations are different to these then you will have to run configure with options specifying the correct paths and the re-run make and sudo make install. That is,

```
./configure -with-R-include-path={path to R include folder}\
            -with-R-lib-path={path to R lib folder}
```

For further information you can run,

```
./configure -help
```

When compiled without error you should have the static libraries *ADlib* and *ADlibErr* installed in */usr/local/lib* and adt installed in */usr/local/bin*. Running,

```
adt
```

should display the help text for ADT.

1.3 Building ADT on Windows

Building ADT on Windows can be done in one of two ways. Firstly, by using a CygWin or MinGW install with Automake utilities installed, adt can be built using the same procedure as for building under Linux.

Alternatively, VC solution and workspace files are included in the *objs* folder and may be used to build adt using Visual C++. If building via Visual C++, it is necessary to install flex and bison and have them in the system path.

A Windows compatible Flex can be found here:

```
http://gnuwin32.sourceforge.net/packages/flex.htm
```

A Windows compatible Bison can be found here:

```
http://gnuwin32.sourceforge.net/packages/bison.htm
```

When building with Visual C++ the executable will be in either the *debug* or *release* folder under the *objs* folder, depending on whether you choose to build a debug or release version. When the build is complete running,

```
./objs/debug/adt
```

or,

```
./objs/release/adt
```

should display the help text for ADT.

1.4 Running ADT

Before you run Adt you must ensure that you have TAPENADE installed and running on your machine. Once TAPENADE is installed and can be run from a shell command prompt, you can run adt in the following manner.

```
adt <makefile1>[ <makefile2> ...]
```

The makefile adheres to the format detailed below. Required user data is enclosed in <>. Optional data is enclosed in [].

Filenames and paths only need to be quoted when whitespace occurs in the name. Any extra arguments passed to TAPENADE using the USER parameter must be quoted in a single string. Single line comments begin with // and free formating of the makefile is allowed provided the syntax is preserved.

The source code classes whose methods are being differentiated must be derrived from the tuseful class and can only use the array types provided by that class. Any other non-scalar types are not allowed in differentiated code.

```
AD
BEGIN
  [SWITCHES: switch1,switch2,...,switchN;] //Optional
  [BLACKBOX: File1,File2,...,FileN;] // Optional
  PATHS: <path 1[,path 2, ... ,path n]>;
  SOURCE FILES: <include file 1[, include file 2, ... ,include file n]>;
  WORKING DIRECTORY: <path to working directory>;

  // language dependent file defining source code macro substitutions and
  // array types - Required.
  PASCAL OPTIONS FILE: pascal_macros.txt;
  // language dependent file defining source code macro substitutions and
  // array types - Required.
  CPP OPTIONS FILE: cpp_macros.txt;

  // file defining fortran functions needed to do the dfferentiation - Required.
  FORTRAN INCLUDE FILES: stdlib.f;

  CLASS <DifferentiatedClass_1> (<ClassBeingDifferentiated_1>)
  SOURCE FILE: <class source file>
  OUTPUT FILE: <differentiated class destination file>
  BEGIN
    FUNCTION=<function_1_to_differentiate> // The function to differentiate
    OUTVAR=<ovar_1[,ovar_2,...,ovar_n]> // Specify the output variables
    VAR=<diff_wrt_var> // Specify the variable to differentiate with respect to
      // Tapenade differentiation modes
      // ---------------
      // f = forward or tangent mode
      // r = reverse mode
      // m = multi-directional tangent mode
    MODE=<Tapenade differentiation mode>
    USER=<'extra tapenade command line args'>; // Additional command line
```

```
        // arguments to pass to Tapenade - these must be bracketed by quotes

      .
      .
      .

      FUNCTION=<function_n_to_differentiate>
      OUTVAR=<ovar_1[,ovar_2,...,ovar_n]>
      VAR=<diff_wrt_var>
      MODE=<Tapenade differentiation mode>
      USER=<'extra tapenade command line args'>;
    END

    .
    .
    .

    CLASS <DifferentiatedClass_n> (<ClassBeingDifferentiated_n>)
    SOURCE FILE: <class source file>
    OUTPUT FILE: <differentiated class destination file>
    BEGIN
      FUNCTION=<function_1_to_differentiate>
      OUTVAR=<ovar_1[,ovar_2,...,ovar_n]>
      VAR=<diff_wrt_var>
      MODE=<Tapenade differentiation mode>
      USER=<'extra tapenade command line args'>;

      .
      .
      .

      FUNCTION=<function_n_to_differentiate>
      OUTVAR=<ovar_1[,ovar_2,...,ovar_n]>
      VAR=<diff_wrt_var>
      MODE=<Tapenade differentiation mode>
      USER=<'extra tapenade command line args'>;
    END
  END
```

Note that in the case of C++ code generation the OUTPUT FILE entry is replaced by OUTPUT FILES and both the destination header and source file names must be specified. That is,

```
     CLASS <DifferentiatedClass_n> (<ClassBeingDifferentiated_n>)
       SOURCE FILE: <class source file>
```

```
    OUTPUT FILES: <differentiated class destination header file>
                  <differentiated class destination source file>
    BEGIN
      FUNCTION=<function_1_to_differentiate>
      OUTVAR=<ovar_1[,ovar_2,...,ovar_n]>
      VAR=<diff_wrt_var>
      MODE=<Tapenade differentiation mode>
      USER=<'extra tapenade command line args'>;

      .
      .
      .

      FUNCTION=<function_n_to_differentiate>
      OUTVAR=<ovar_1[,ovar_2,...,ovar_n]>
      VAR=<diff_wrt_var>
      MODE=<Tapenade differentiation mode>
      USER=<'extra tapenade command line args'>;
    END
  END
```

Chapter 2

Using ADT

2.1 An R auto-differentiation example

A complete auto-differentiation sample application can be found in the folder *sample*. This folder contains three sub folders : *ArrayTest*, *common*, *include* and *Rosenbrock*.

ArrayTest contains a test harness application built using *ADLib* and the automation provided by ADT.

common contains useful Pascal source code.

include contains files needed by ADT to correctly function.

Rosenbrock contains files needed to build and run the example R dynamic library, which in this case, is a simple minimisation of the Rosenbrock minimisation problem for testing the convergence of minimisers.

If all compiled without error when building ADT, then you should already have a complete shared library for this example. To run the example change directories to the *Rosenbrock* folder. For example,

```
cd sample/Rosenbrock
```

now start R with,

```
R
```

This assumes that R is in the executable search PATH. If it isn't then you will need to fully qualify the command.

In the R session enter the following command,

```
source('Rosenbrock.R', print.eval=TRUE)
```

On doing so and after a minute or so you should see some results displayed similar to the following:

```
[1] "Comparison of minimisation run times and performance for different "
[2] "implementations of the Rosenbrock minimisation problem with "
[3] "150 parameters. Minimisation performed using nlminb. "
                                                   case objective    RMS.error
1                                           All R based         0 7.307681e-12
2                                         C++ objective         0 1.505906e-09
3                             C++ objective - transposing         0 1.505906e-09
4                              C++ objective and gradient         0 5.016338e-12
5                C++ objective and gradient - transposing         0 5.016338e-12
6                     C++ objective, gradient and hessian         0 0.000000e+00
7 C++ objective, gradient and hessian - transposing         0 0.000000e+00
  convergence time.taken iterations fn.evaluations grad.evaluations
1           0      43.12        501            818            83418
2           1       0.73        321            539            68260
3           1       0.76        321            539            68260
4           1       0.65        496            856              496
5           1       0.66        496            856              496
6           0       0.43         18             29               19
7           0       9.17        255            347              256
              message
1     X-convergence (3)
2 false convergence (8)
3 false convergence (8)
4 false convergence (8)
5 false convergence (8)
6     X-convergence (3)
7     X-convergence (3)
```

This shows the comparative performance of various minimisations of the same Rosenbrock function, implemented in R, in C++ and with an without algorithmically differentiated gradient and hessian. Having access to machine precision gradient and hessian has a significant effect on minimiser performance. In particular, it is interesting to note that with machine precision gradient and hessian *nlminb* was able to find the solution exactly and in the shortest amount of time.

Study the example code, the adt makefile and the Tapenade documentation to understand how it works and how to solve your own AD problems.

2.2 The example in more detail

A Rosenbrock function is a non-convex function of know minimum used to test the convergence performance of minimisation algorithms and was introduce by Howard H. Rosenbrock in 1960. For the two dimensional form the global minimum is situated in a long parabolic shaped valley. To find the valley is easy but finding the minimum is significantly harder.

The two dimensional form is defined as,

$f(x,y) = (1-x)^2 + 100\left(y-x^2\right)^2$

and has a global mimimum at $(x,y) = (1,1)$of $f(x,y) = 0$.

The multi-dimensional form is defined as,

$$f(X) = f(x_1, x_2, ..., x_N) = \sum_{i=1}^{N/2}\left[100\left(x_{2i-1}^2 - x_{2i}\right)^2 + (x_{2i-1}-1)^2\right]$$

where N is even. This is the form we shall use to test the convergence of the *nlminb()* minimiser in R.

To implement a minimiser test we need to first implement the above function which we shall do in C++. ADT only supports parsing a subset of the C++ language. The limitations on C++ use are as follows:

- Only intrinsic types and the array types defined in *AdtArrays.hpp* may be used in your code.
- Classes cannot be used in code that is to be automatically differentiated.
- Local variable declarations must be in one block at the beginning of a function definition (much like C syntax) and combined declaration - initialisation is not allowed.
- Use of case to distinguish between variables is not allowed. This is because the AD part carried out by Tapenade is in Fortran code which is case insensitive. Any discrimination by case is lost in the translation to Fortran.
- Any function to be automatically differentiated must be a public method of a class publically derived from *AdtArray*.
- Function overloading for differentiated code is not allowed (that is providing multiple implementations of the same function that differ only by the function argument list).
- Meta data is required in the form of comments to instruct ADT about the size of arrays and for the automatic creation of R interface code (see section 3.3 for more details).

With that in mind we implement a simple class called *MinRosenbrock* (in files *Rosenbrock.hpp* and *Rosenbrock.cpp*) as shown below.

```
#include <adtarrays.hpp>
#include "adtR.hpp"

// -------------------------------

class MinRosenbrock : public AdtArrays
{
  protected:
   /* AD_LIBNAME Rosenbrock */
   /* AD_ALIAS Rb=D_MinRosenbrock */
   /* AUTOINIT */
   int     N;

#include "Rb_array_plans.hpp"

public:
   MinRosenbrock(
#include "Rb_constructor_args.hpp"
   );

#include "Rb_decl_lib_interface_methods.hpp"

  double  rosenbrock(const ARRAY_1D X/*N*/);
};
```

Firstly, it includes the files *adtarrays.hpp* (needed for the array support) and *adtR.hpp* (needed for the R interface support). Secondly, it declares the class *MinRosenbrock*, which is derived from *AdtArrays*, and declares and implements the method *rosenbrock()* and the attribute *N*, both used to implement our N order rosenbrock function. Thirdly, it has a bunch of comments to instruct ADT how to process the file.

The comments,

```
/* AD_LIBNAME Rosenbrock */
/* AD_ALIAS Rb=D_MinRosenbrock */
/* AUTOINIT */
```

instruct ADT that the library name to be generated is Rosenbrock (*AD_ LIBNAME* command), instruct ADT to generate R interface code for this class (*AD_ ALIAS* command) and automatically initialise the N attribute (*AUTOINIT* command). The comment in,

```
double  rosenbrock(const ARRAY_1D X/*N*/);
```

instructs ADT that the array *X* has a dimension of *N*. The other include file directives are includes inserted to include the automatically generated R interface and intialisation code.

In *Rosenbrock.cpp* the class is implemented as is shown below.

```
#include "Rosenbrock.hpp"

// -------------------------------

MinRosenbrock::MinRosenbrock(
#include "Rb_constructor_args.hpp"
)
{
  #include "Rb_constructor_locals.hpp"
  #include "Rb_constructor_scalars_phase_1.hpp"
  #include "Rb_constructor_arrays_phase_1.hpp"
  #include "Rb_array_plans_init.hpp"
}

// ------------------------------

double MinRosenbrock::rosenbrock(const ARRAY_1D X/*N*/)
{
  int cn;
  double dSum;
  double p1,p2,f1,f2;

  dSum = 0.0;

  for (cn = 1 ; cn <= N / 2 ; cn++)
  {
    p1 = X[2 * cn - 1];
    p2 = X[2 * cn];
    f1 = (p1 * p1 - p2);
    f1 = 100 * f1 * f1;
    f2 = (p1 - 1.0);
    f2 = f2 * f2;
    dSum += f1 + f2;
  }

  return (dSum);
}
```

Note again, the presence of include directives for the automatically generated code. From inspection it should be seen that the method *rosenbrock()* does indeed implement the Rosenbrock function as defined earlier.

Having defined the method and its containing class an ADT make file is required to instruct ADT on which files to process and which functions to differentiate. This is the purpose of the file *Rosenbrock.mk* shown below (see section 3.2 for more information on ADT make files).

```
AD BEGIN
  PATHS: "../include","../common", "../../src/ADLib";
  WORKING DIRECTORY: "./work/";
  CPP OPTIONS FILE: cpp_macros.txt;
  PASCAL OPTIONS FILE: pascal_macros.txt;
  FORTRAN INCLUDE FILES: stdlib.f;
```

```
    CLASS D_MinRosenbrock(MinRosenbrock)
      SOURCE FILE: Rosenbrock.cpp
      OUTPUT FILES: d_Rosenbrock.cpp d_Rosenbrock.hpp
    BEGIN
      FUNCTION=rosenbrock OUTVAR=rosenbrock VAR=X MODE=r;
      FUNCTION=ROSENBROCK_BX OUTVAR=xb1_x VAR=X MODE=f;
    END
    CLASS DR_MinRosenbrock(R_MinRosenbrock)
      SOURCE FILE: R_Rosenbrock.cpp
      OUTPUT FILES: dR_Rosenbrock.cpp dR_Rosenbrock.hpp
    BEGIN
    END
  END
```

Each class entry instructs ADT to build a new class derived from the one listed and apply the given AD operations to the specified *FUNCTIONs*. In this case that class is *MinRosenbrock* and the method to differentiate is *rosenbrock* (to give the gradient function). It is differentiated in reverse mode (r) to obtain the gradient vector rather than forward mode (f) which would only give a directional derivative. As a hessian implementation is also required we also take the directional derivative of *ROSENBROCK_BX*, the gradient function that resulted from the previous AD operation.

The hessian function must be hand written using the directional second derivative. These functions are implemented in the class *R_MinRosenbrock* (found in *R_Rosenbrock.cpp* and *R_Rosenbrock.hpp*). Since ADT will create R code to instantiate the differentiated version of the class, a null differentiation step is required so that the instantiated class will be a parent of *R_MinRosenbrock*, hence the final null AD step in the makefile. It is possible to leave that step out but doing so would mean that the R interface code would need to be hand written, which is a tedious and error prone process.

ADT also automatically creates R interface code for all public methods of every AD class and getter and setter methods to access automation initialised attributes of those classes.

2.3 An R auto-differentiation example - Pascal version

If you wish to code in Pascal rather than C++ a Pascal sample of the same problem is also included.

Before you can run the Pascal version you must first build the shared library. Under Windows run the batch file *build_lib.bat*. Under Linux distributions run *build_lib.sh*.

To run the Pascal example change directories to the *Rosenbrock* folder. For example,

```
cd sample/Rosenbrock
```

now start R with,

```
R
```

This assumes that R is in the executable search PATH. If it isn't then you will need to fully qualify the command.

In the R session enter the following command,

```
source('RosenbrockPas.R', print.eval=TRUE)
```

On doing so and after a minute or so you should see some results displayed similar to the following:

```
[1] "Comparison of minimisation run times and performance for different "
[2] "implementations of the Rosenbrock minimisation problem with "
[3] "150 parameters. Minimisation performed using nlminb. "
                                                case objective    RMS.error
1                                         All R based         0 7.307681e-12
2                                    Pascal objective         0 2.354987e-11
3                       Pascal objective - transposing         0 2.354987e-11
4                        Pascal objective and gradient         0 1.473745e-11
5          Pascal objective and gradient - transposing         0 1.473745e-11
6               Pascal objective, gradient and hessian         0 0.000000e+00
7 Pascal objective, gradient and hessian - transposing         0 5.933984e-14
  convergence time.taken iterations fn.evaluations grad.evaluations
1           0      42.62        501            818            83418
2           1       1.47        574            938            98685
3           1       1.58        574            938            98685
4           1       0.67        494            824              494
5           1       0.70        494            824              494
6           0       0.49         18             29               19
7           0       9.91        257            355              258
              message
1     X-convergence (3)
2 false convergence (8)
3 false convergence (8)
4 false convergence (8)
5 false convergence (8)
6     X-convergence (3)
7     X-convergence (3)
```

This shows the comparative performance of various minimisations of the same Rosenbrock function, implemented in R, in Pascal and with an without algorithmically differentiated gradient and hessian. Having access to machine precision gradient and hessian has a significant effect on minimiser performance. In particular, it is interesting to note that with machine precision gradient and hessian *nlminb* was able to find the solution exactly and in the shortest amount of time.

Chapter 3

ADT Reference

3.1 TAPENADE Primer

TAPENADE is both a command line and a web based automatic / algorithmic differentiation engine that takes source code in Fortran and differentiates the specified functions to produce modified Fortran code which calculating the differentials. This is loosely termed as AD by code translation as opposed to AD by operator overloading. The latter exploits the language features of object oriented languages like C++ to implement AD and generally produces code that is typically not as fast as that through the code translation approach. The added efficiencies available in the latter stem from the ability to be able to optimise out common subexpressions so that they are evaluated only once, which is something very difficult to achieve automatically in the operator overloading approach. ADT makes use of the command line version of TAPENADE to synthesise differentials.

For reasons of efficiency TAPENADE does not produce differentials directly. Instead it produces them in two common flavours, *Tangent* codes (Jacobian times Vector) or *Forward* mode and *Adjoint* codes (Transposed Jacobian times Vector) or *Reverse* mode. Several strategies are employed in TAPENADE to produce efficient codes. TAPENADE calculates differentials of scalar functions. Should the output of a function be a vector quantity then TAPENADE interperates it as a system of scalar equations and differentiates each individually, returning a vector quantity of the same dimension to represent the result.

3.1.1 Forward Mode

Forward or Tangent mode calculates the sensitivities of the output variables to the input variables. For each output variable TAPENADE creates a new output variable with the same name but with *d* appended to it (d for “dot”). Similarly for each input variable it creates a new input variable with the same name but with *d* appended to it. These input variables are multiplied with the corresponding differential in a dot product with the Jacobian. To illustrate, here is a simple contrived example. Our Fortran function source code is,

```
DOUBLE PRECISION FUNCTION trig(x, y)
  INTENT (IN) x, y
  trig = cos(3 * x) + sin(y / 5)
  RETURN
END
```

Running it through TAPENADE specifying trig as the head routine and asking for a tangent mode output, TAPENADE responds with,

```
!        Generated by TAPENADE     (INRIA, Tropics team)
!  Tapenade 3.7 (r4888) - 28 May 2013 10:47
!
!  Differentiation of trig in forward (tangent) mode:
!   variations   of useful results: trig
!   with respect to varying inputs: x y
!   RW status of diff variables: x:in y:in trig:out
DOUBLE PRECISION FUNCTION TRIG_D(x, xd, y, yd, trig)
  IMPLICIT NONE
  REAL :: x
  REAL :: y
  INTENT(IN) x, y
  REAL :: xd
  REAL :: yd
  INTENT(IN) xd, yd
  DOUBLE PRECISION :: trig
  INTRINSIC COS
  INTRINSIC SIN
  trig_d = yd*COS(y/5)/5 - 3*xd*SIN(3*x)
  trig = COS(3*x) + SIN(y/5)
  RETURN
END FUNCTION TRIG_D
```

Studying the code you can see that the differential output *trig_ d* is the dot product of the Jacobian with the vector [*xd, yd*]. In addition to returning this sensitivity result, it also returns the result of the original function evaluated at the same point in the new variable *trig*. This is the way TAPENADE behaves when used in tangent / foward mode.

From the above it should be apparent that we can obtain the partial differentials for the function by specifying the correct values for *xd* and *yd*. For example,

```
x_dot = TRIG_D(x, 1, y, 0, trig)
```

will return the partial differential of trig at *x,y* with respect to *x* and,

```
y_dot = TRIG_D(x, 0, y, 1, trig)
```

will return the partial differential of trig at *x,y* with respect to *y*.

3.1.2 Reverse Mode

Reverse or Adjoint mode calculates the scaled gradient vector for the output variables with respect to the dependent input variables. For each input variable TAPENADE creates a new output variable with the same name but with *b* appended to it (b for "bar"). Similarly for each output variable it creates a new input variable with the same name but with *b* appended to it. Applying TAPENADE to the same source code as the Forward Mode example but requesting a adjoint / reverse mode output, TAPENADE gives,

```
!        Generated by TAPENADE     (INRIA, Tropics team)
!  Tapenade 3.7 (r4888) - 28 May 2013 10:47
!
!  Differentiation of trig in reverse (adjoint) mode:
!   gradient     of useful results: x y trig
!   with respect to varying inputs: x y
!   RW status of diff variables: x:incr y:incr trig:in-killed
SUBROUTINE TRIG_B(x, xb, y, yb, trigb)
  IMPLICIT NONE
  REAL :: x
  REAL :: y
  INTENT(IN) x, y
  DOUBLE PRECISION :: trigb
  DOUBLE PRECISION :: trig
  REAL :: xb
  REAL :: yb
  INTRINSIC COS
  INTRINSIC SIN
  xb = xb - SIN(3*x)*3*trigb
  yb = yb + COS(y/5)*trigb/5
END SUBROUTINE TRIG_B
```

from which it is clear that *TRIG_B* returns the gradient result for the function when *trigb* is set to one and *xb* and *yb* are intialised to zero before calling *TRIG_B*. Note that the rules governing the introduction of *xb*, *yb* and *trigb* and their processing may look odd but is done this way to facilitate the chaining of operations through application of the chain rule to obtain the adjoint differential of a given outer function.

Reverse mode is an efficient means of finding the gradient (partial differentials with respect to all dependent inputs) but it requires a tape / stack to store partial results on the reverse parse. The simplicity of the previous example meant no stack was required but in general one is and TAPENADE will insert

PUSHes and POPs to save and restore partial results. Should you wish to apply second order differentiation to a reverse mode output the PUSHes and POPs must be removed and replaced with arrays. ADT does exactly that.

PUSHing and POPing everything to a stack can consume vast amounts of memory for complex problems, which in turn can make computation time baloon out as memory resources are exhausted. To combat this problem TAPENADE uses checkpointing, which in laymans terms, approximately means breaking the problem up into smaller pieces to reduce the stack requirements. TAPENADE does checkpointing on a function / subroutine call basis so that the stack requirements of function calls within a function are isolated from the enclosing function.

3.1.3 ADT Naming Convention

The naming convention used for the creation of new variables and functions by TAPENADE is an extension of the TAPENADE default behaviour. Since the primary aim of ADT is to easily allow higher order differentiation, even when using reverse mode, we need to create new variables in a manner that will not result in name clashes with previous output. As such, the naming convention in ADT includes an iteration number that corresponds to the number of the operation in the list of FUNCTIONs to be AD'ed within the ADT makefile and the variables we are differentiating with respect to.

The variable and function naming conforms to the following templates.

FUNCTION

Forward Mode	`{function name}_D{dependent variables}`
Reverse Mode	`{function name}_B{dependent variables}`

VARIABLE

Forward Mode	`{variable name}d{iteration}_{dependent variables}`
Reverse Mode	`{variable name}b{iteration}_{dependent variables}`

As an example, consider the makefile from the demonstration project in ADT as shown below,

```
AD BEGIN
  PATHS: "../include","../common", "../../src/ADLib";
  WORKING DIRECTORY: "./work/";
  CPP OPTIONS FILE: cpp_macros.txt;
  PASCAL OPTIONS FILE: pascal_macros.txt;
  FORTRAN INCLUDE FILES: stdlib.f;

  CLASS D_MinRosenbrock(MinRosenbrock)
    SOURCE FILE: Rosenbrock.cpp
    OUTPUT FILES: d_Rosenbrock.cpp d_Rosenbrock.hpp
  BEGIN
```

```
      FUNCTION=rosenbrock OUTVAR=rosenbrock VAR=X MODE=r;
      FUNCTION=ROSENBROCK_BX OUTVAR=xb1_x VAR=X MODE=f;
    END

    CLASS DR_MinRosenbrock(R_MinRosenbrock)
      SOURCE FILE: R_Rosenbrock.cpp
      OUTPUT FILES: dR_Rosenbrock.cpp dR_Rosenbrock.hpp
    BEGIN
    END
  END
```

The first AD operation is reverse mode differentiation with respect to X of the function *rosenbrock*. Therefore TAPENADE produces the differentiated function *ROSENBROCK_BX* with the output gradient variable *xb1_x* and the input scaling value *rosenbrockb1_x* because the dependent variable is x and the iteration number is 1. The second AD operation is forward mode differentiation with respect to x of *ab1_x* of *ROSENBROCK_BX* which yields the function *ROSENBROCK_BX_DX* with output variable *xb1_xd2_x* and input variable *xd2_x* because the dependent variable is x and the iteration number is 2.

3.2 ADT Make Files

3.2.1 Structure

The text within $< >$ is user supplied data, the text within [ ] is optional, the remainder is necessary text required to satisfy the make file grammar.

```
  AD
  BEGIN
    [SWITCHES: switch1,switch2,...,switchN;] //Optional
    [BLACKBOX: File1,File2,...,FileN;] // Optional
    PATHS: <path 1[,path 2, ... ,path n]>;
    [SOURCE FILES: <include file 1[, include file 2, ... ,include file n]>;]
    WORKING DIRECTORY: <path to working directory>;
    PASCAL OPTIONS FILE: <pascal options file>;
    CPP OPTIONS FILE: <C++ options file>;
    FORTRAN INCLUDE FILES: <fortran include file>
    [SWITCHES <switch 1,switch 2, ... ,switch n>];

    CLASS <DifferentiatedClass_1> (<ClassBeingDifferentiated_1>)
    SOURCE FILE: <class source file>
    OUTPUT FILE: <differentiated class destination file>
    BEGIN
      [BOUNDS CHECK fn1, fn2,...,fnN;] //Optional
      FUNCTION=<function_1_to_differentiate>
    OUTVAR=<ovar_1[,ovar_2,...,ovar_n]>
    VAR=<diff_wrt_var_1[,diff_wrt_var_2,...,diff_wrt_var_n]>
    MODE=<Tapenade diffMake File erentiation mode>
    USER=<'extra tapenade command line args'>
    [PRAGMAS=<'Pragma options text'>];

    .
    .
    .

    FUNCTION=<function_n_to_differentiate>
    OUTVAR=<ovar_1[,ovar_2,...,ovar_n]>
    VAR=<diff_wrt_var_1[,diff_wrt_var_2,...,diff_wrt_var_n]>
    MODE=<Tapenade differentiation mode>
    USER=<'extra tapenade command line args'>
    [PRAGMAS=<'Pragma options text'>];
  END

  .
  .
  .

  CLASS <DifferentiatedClass_n> (<ClassBeingDifferentiated_n>)
  SOURCE FILE: <class source file>
  OUTPUT FILE: <differentiated class destination file>
  BEGIN
    [BOUNDS CHECK fn1, fn2,...,fnN;] //Optional
    FUNCTION=<function_1_to_differentiate>
    OUTVAR=<ovar_1[,ovar_2,...,ovar_n]>
    VAR=<diff_wrt_var_1[,diff_wrt_var_2,...,diff_wrt_var_n]>
    MODE=<Tapenade differentiation mode>
    USER=<'extra tapenade command line args'>
    [PRAGMAS=<'Pragma options text'>];

    .
    .
    .

    FUNCTION=<function_n_to_differentiate>
    OUTVAR=<ovar_1[,ovar_2,...,ovar_n]>
    VAR=<diff_wrt_var_1[,diff_wrt_var_2,...,diff_wrt_var_n]>
    MODE=<Tapenade differentiation mode>
    USER=<'extra tapenade command line args'>
    [PRAGMAS=<'Pragma options text'>];
  END
END
```

3.2.2 Incremental Compilation

As the number of AD operations and AD classes grow, the time taken to generate or compile the AD code grows too. To minimise the compilation time during the developmental phase of creating a library with ADT, ADT includes an incremental compilation mechanism whereby it only re-compiles what it thinks it has to based on a dependency checking protocol. That protocol takes the source file locations and dates and the operations carried out on each class and class methods as an indicator of whether re-compilation is necessary.

If any of the source files, working directory, pascal and C++ options files, fortran include files and switches change then the entire project is re-built. If the class file changes then the entire class and any subsequent classes are re-built. The implicit assumption in the design is that any CLASSes appearing in the make after this one depend on it, so a change in the first one invalidates the remainder. The individual AD operations defined in each of the FUNCTION entries are checked for a change. If no change is detected then the cached build from a previous build is used. If a change is detected then the AD operations are re-envoked and everything that follows is as well. In this way, adding new functions one at a time will only result in a compilation and AD operation step for the added function, the remainder is cached.

The dependency information used to check if a re-build is required is stored in a file with *.dch* extension, the name corresponding to the name of the make file less the file extension. Therefore, to force a full re-compile in the case where you suspect incremental compilation as a problem, you can simply delete the *.dch* file to bring about that outcome.

3.2.3 Entries in Detail

3.2.3.1 PATHS

```
PATHS: <path 1[,path 2, ... ,path n]>;
```

The PATHS entry specifies the folder paths to be used when searching for the source files referenced in the make file.

3.2.3.2 SOURCE FILES

```
[SOURCE FILES: <include file 1[, include file 2, ... ,include file n]>;]
```

The SOURCE FILES entry names any additional source files needed to carry out automatic differentiation of functions specified in the CLASS sections. This generally means the source files containing any code being called by the function being differentiated that does not reside in the same source file as the function itself. You should only reference files containing differentiable code here. The source files named must be accessible through one of the paths specified in

PATHS. If your project has no such external file dependencies then the SOURCE FILES entry can be completely omitted.

3.2.3.3 WORKING DIRECTORY

```
WORKING DIRECTORY: <path to working directory>;
```

The WORKING DIRECTORY entry specifies the PATH to the folder in which ADT puts working files, which are intermediate files created to carry out the automatic differentiation operations requested. Having access to these working files is of particular use when trying to debug the automatic differentiation process. TAPENADE can sometimes create incorrect code and when this happens it is necessary to study the inputs and outputs of TAPENADE to determine the cause of the problem and remedy it.

3.2.3.4 PASCAL OPTIONS FILE

```
PASCAL OPTIONS FILE: <PASCAL options file>;
```

The PASCAL OPTIONS FILE entry names the file containing options controlling the interpretation and translation of Pascal code into FORTRAN and back again. You will typically use the pascal_macros.txt supplied with ADT and provide a path to it. That is,

```
PASCAL OPTIONS FILE: pascal_macros.txt;
```

3.2.3.5 CPP OPTIONS FILE

```
CPP OPTIONS FILE: <C++ options file>;
```

The CPP OPTIONS FILE entry names the file containing options controlling the interpretation and translation of C++ code into FORTRAN and back again. You will typically use the cpp_macros.txt supplied with ADT and provide a path to it. That is,

```
CPP OPTIONS FILE: cpp_macros.txt;
```

3.2.3.6 FORTRAN INCLUDE FILES

```
FORTRAN INCLUDE FILES: <include file 1[, include file 2, ... ,include file n]>;
```

The FORTRAN INCLUDE FILES entry names any FORTRAN source files needed to carry out automatic differentiation of functions specified in the CLASS sections. This is provided to give an opportunity to introduce function stubs for black box routines. You will typically use the stdlib.f file supplied with ADT and provide a path to it. That is,

```
FORTRAN INCLUDE FILES: stdlib.f;
```

3.2.3.7 SWITCHES

```
[SWITCHES <switch 1,switch 2, ... ,switch n>];
```

The SWITCHES entry lists any control switches to alter the normal behaviour of ADT in processing the make file. Currently the following switches are defined.

BoundsCheck Enables the bounds checking of all functions defined in the parent class and the AD result class. See section 3.2.3.10 for more information on bounds checking.

WithoutStackSubstitution Disables the replacement of PUSH and POP calls with arrays. This is principally for debugging ADT behaviour.

ThrowException Enables the throwing of an exception on bounds check failures. If not defined no exception is thrown but the error is reported. In an R context the R stop() function is called.

3.2.3.8 BLACKBOX

```
[BLACKBOX: <black box file 1[, black box file 2, ... ,black box file n]>;]
```

The BLACKBOX entry names any additional black box definition files to be passed to Tapenande when carrying out algorithmic differentiation operations. These files conform to the syntax for black box definitions in ADT (see section 3.3.2.6) and are compiled and translated into the more complex form required by TAPENADE. All black box definitions are combined to form a single black box definition file that is passed to TAPENADE through the `-ext` command lin option.

3.2.3.9 CLASS

```
 CLASS <DifferentiatedClass> (<ClassBeingDifferentiated>)
    SOURCE FILE: <class source file>
    OUTPUT FILE: <differentiated class destination file>

 CLASS <DifferentiatedClass> (<ClassBeingDifferentiated>)
    SOURCE FILES: <class source file> <class header file>
    OUTPUT FILES: <differentiated class destination file>
<differentiated class header file>
```

The CLASS entry names the class (DifferentiatedClass) resulting from the application of the AD operations and the class (ClassBeingDifferentiated) whose methods are being AD'd. The implementation of the class being differentiated must reside in the SOURCE FILE and the resulting differentiated code will reside in the OUTPUT FILE. In a C++ context you may specify a source file and source file header using the SOURCE FILES form. Similarly, you may specify an output source file and header file using the OUPUT FILES form. The source files named must be accessible through one of the paths specified in PATHS.

3.2.3.10 BOUNDS CHECK

```
BOUNDS CHECK fn1, fn2,...,fnN;
```

The BOUNDS CHECK entry names the functions whose enclosed code array indexing will be bound checked for over / under-runs. Functions named in this list need not be pre-existing but should exist as a result of any subsequent AD operations. Naming non-existent functions will not flag as an error.

Any bounds checked functions in the parent class are re-implemented in the differentiated class with bounds checking code included. For parent class overrides the methods must be declared as virtual to ensure that the bounds checked version is called.

3.2.3.11 FUNCTION

```
FUNCTION=<function_to_differentiate>
    OUTVAR=<ovar_1[,ovar_2,...,ovar_n]>
    VAR=<diff_wrt_var_1[,diff_wrt_var_2,...,diff_wrt_var_n]>
    MODE=<Tapenade differentiation mode>
    USER=<'extra tapenade command line args'>
    [PRAGMAS=<'Pragma options text'>];
```

The FUNCTION entry names the class function / method to be differentiated (function_to_differentiate).

The output variable/variables are named by the OUTVAR parameter. In the case of an output variable being returned by the function / method, it is given the same name as the function. The variable/variables to differentiate with respect to is/are named by the VAR parameter. Names are comma seperated and case insensitive.

The MODE parameter specifies the mode of automatic differentiation which can be one of ,

f forward or tangent mode

r reverse or adjoint mode

m multi-directional tangent mode

The USER parameter specifies any addition arguments to pass to the command line call to TAPENADE. This is a string that pasted verbatim into the TAPENADE command line. You will typically specify the black box definitions supplied with ADT in this argument with the line,

```
USER='-ext ../include/my_extlib.txt'
```

The PRAGMAS option provides a means of specifying switches to alter the behaviour of ADT in performing the AD operations.Currently the following pragmas are defined.

PushPopDisable Disables the pushing and popping of arrays onto the stack / tape in reverse mode differentiation

WithoutStackSubstitution Disables the replacement of PUSH and POP calls with arrays. This is principally for debugging ADT behaviour.

WithStackSubstitution Enables the replacement of PUSH and POP calls with arrays. This is principally for debugging ADT behaviour.

3.2.4 Make Grammar

Keywords

```
AD BEGIN BOUNDS CHECK CLASS DIRECTORY PASCAL END FILE FILES FORTRAN
FUNCTION INCLUDE MODE OPTIONS OUTPUT OUTVAR PATHS PRAGMAS
SOURCE SWITCHES USER VAR WORKING BLACKBOX
```

Tokens

```
FileName:
   A valid file path either quoted or non-quoted.  Either single or
   double quote characters may be used.

Ident:
   An identifier starting with a non-numeric character and containing
   only numbers, letters and underscore characters.

Text:
   A string of characters on a single line enclosed in single quotes.
```

Rules

```
MakeCommandList:
   MakeCommand
   MakeCommandList MakeCommand

MakeCommand:
   AD BEGIN MakeOptionsList ClassList END

MakeOptionsList:
   MakeOption
   MakeOptionsList MakeOption

MakeOption:
   PATHS ':' FileNameList ';'
   SOURCE FILES ':' FileNameList ';'
   WORKING DIRECTORY ':' FileName ';'
   CPP OPTIONS FILE ':' FileName ';'
   PASCAL OPTIONS FILE ':' FileName ';'
   FORTRAN INCLUDE FILES ':' FileNameList ';'
   SWITCHES ':' FileNameList ';'
   BLACKBOX ':' FileNameList ';'

FileNameList:
   FileName
   FileNameList ',' FileName

ClassList:
   ClassBlock
   ClassList ClassBlock

ClassBlock:
   ClassBlockBegin AutoDiffCommandList END
   ClassBlockBegin END

ClassBlockBegin:
   CLASS Ident '(' Ident ')' SOURCE FILE ':' FileName OUTPUT FILE
   ':' FileName BEGIN
   CLASS Ident '(' Ident ')' SOURCE FILE ':' FileName OUTPUT FILES
   ':' FileName FileName BEGIN
   CLASS Ident '(' Ident ')' SOURCE FILES ':' FileName FileName OUTPUT
   FILE ':' FileName BEGIN
   CLASS Ident '(' Ident ')' SOURCE FILES ':' FileName FileName OUTPUT
   FILES ':' FileName FileName BEGIN

AutoDiffCommandList:
   AutoDiffCommand
   AutoDiffCommandList AutoDiffCommand
   AutoDiffCommand:AutoDiffCommandOpList ';'

AutoDiffCommandOpList:
   AutoDiffCommandOp
   AutoDiffCommandOpList AutoDiffCommandOp

AutoDiffCommandOp:
   AutoDiffCommandFunctionOp
   AutoDiffCommandVarOp
   AutoDiffCommandOutVarOp
   AutoDiffCommandModeOp
   AutoDiffCommandUserOp
   AutoDiffCommandPragmasOp
   AutoDiffCommandBoundsCheckOp

AutoDiffCommandFunctionOp:
   FUNCTION '=' Ident

AutoDiffCommandVarOp:
   VAR '=' NameList

AutoDiffCommandOutVarOp:
   OUTVAR '=' NameList

AutoDiffCommandModeOp:
   MODE '=' Ident

AutoDiffCommandUserOp:
   USER '=' Text

AutoDiffCommandPragmasOp:
   PRAGMAS '=' Text

AutoDiffCommandBoundsCheckOp:
   M_BOUNDS M_CHECK NameList ';'

NameList:
   Ident
   NameList ',' Ident
```

3.3 ADT Automatic R Interface Code

Writing scientific code that interfaces with R is a complex an error prone task. To simplify this process ADT provides a means to automatically create most of the interface code for you. This feature is controlled by the inclusion of comments in the source code that form automation commands which ADT then interprets to produce the desired code.

3.3.1 Array Dimension Comments

ADT needs to know the sizes of arrays in a project class to both correctly carry out AD operations and to produce correct interface code. We tell ADT the array sizes through inline comments appearing immediately after the declaration of the array in question. The required comment syntax is[1],

```
/*[lower bound index 1:]upper bound index 1,
  [lower bound index 2:]upper bound index 2, ...,
  [lower bound index n:]upper bound index n*/
```

for C++ code and,

```
{[lower bound index 1:]upper bound index 1,
 [lower bound index 2:]upper bound index 2, ...,
 [lower bound index n:]upper bound index n}
```

for Pascal code. The lower bound index is optional and if omitted is assumed to be 1. For example,

```
ARRAY_2D   my_array_a/*m,n*/;
ARRAY_2D   my_array_b/*4:p,-1:q+1*/;
```

in C++ and,

```
my_array_a{m,n}        : ARRAY_2D;
my_array_b{4:p,-1:q+1} : ARRAY_2D;
```

in Pascal, both declare two arrays of doubles: *my_ array_ a* whose first index is between 1 and m and whose second index is between 1 and n, and *my_ array_ b* whose first index is between 4 and p and whose second index is between -1 and q+1. Any variable index limit must be defined as an attribute of the class.

Ragged array definitions are also possible through specifying the ragged dimensions with a vector of indices. For instance,

```
ARRAY_2L   n_obs_y_r /*n_year,n_region*/;
ARRAY_3D   obs /*n_region,n_year,n_obs_y_r[<2>,<1>]*/;
ARRAY_3D   alt_obs /*n_year,n_region,n_obs_y_r[<-2>,<-1>]*/;
```

in C++ and,

```
n_obs_y_r {n_year,n_region}                      : ARRAY_2L;
obs {n_region,n_year,n_obs_y_r[<2>,<1>]}         : ARRAY_3D;
alt_obs {n_year,n_region,n_obs_y_r[<-2>,<-1>]} : ARRAY_3D;
```

in Pascal, define two ragged arrays *obs* and *alt_ obs* whose third dimension is ragged and specified by the array *n_ obs_ y_ r*. Any ragged array indice will be distinguishable from a non-ragged one by the presence of an indexing specification that determines which indices of the size array are driven by which indicies of the ragged array. In the example above for the array *obs* the third index size (`n_obs_y_r[<2>,<1>]`) is obtained from *n_ obs_ y_ r* using *n_ year*

[1]The square brackets enclose optional parameters

(implied by `<2>` which instructs adt to use the second index of *obs* to index into *n_obs_y_r*) as the first index and *n_region* as the second one (implied by `<1>` which instructs adt to use the first index of *obs* to index into *n_obs_y_r*). For the case of *alt_obs* negative / relative indexing is used. For negative indexing the datum is relative to the index from where the specification occurs so the `<-2>` means the index two places to the left of this one or the *n_year* index. In positive indexing the datum is absolute and is the first index of the array being specified, thus the `<1>` refers to the *n_region* index.

Array dimension comments are also used to declared the dimension of arguments in functions and subroutines but in this context ragged array dimensioning is not supported. For example,

```
double my_func(int ix,
               int n,
               const ARRAY_1D array_a /*n*/,
               ARRAY_1D array_b /*m*/);
```

in C++ or,

```
FUNCTION my_func(ix, n: LONGINT;
                 CONST array_a {n}, array_b {m}: ARRAY_1D): DOUBLE;
```

in Pascal.

3.3.2 Preprocessor Comment Commands

By providing preprocessor commands in code comments ADT can automatically generate R interface code to interface your library with R. Every class which is to have automatically generated code must have an AD_ALIAS command appearing as the first command in the class definition.

Class attributes are either automatically intialised, or null initialised or not declared or initialised depending on whether they are define in the scope of AUTOINIT, AUTODEC and MANUAL commands. Furthermore, the timing of the initialisation operations can be controlled with a phase parameter applied in these commands.

3.3.2.1 AD_LIBNAME command

If a library is to interface with R and have all interface routines registered with R when the library is loaded you must tell ADT the name of the library. This is done using the AD_LIBNAME command whose general form is,

```
AD_LIBNAME Library_Name
```

This command can appear anywhere in the class definition.

3.3.2.2 AD_ALIAS command

Every class that is to have an interface to R and have automatically generated constructor code must provide and AD_ALIAS command. In fact if no AD_ALIAS command is found no further preprocessing is performed. The AD_ALIAS command must be the first command the preprocessor reads. The general form of the command is,

```
AD_ALIAS Alias_Name=D_Class_Name[, Parent_Class_Name]
```

where the *Parent_Class_Name* parameter is optional and only used when a class derives from a class that has an AD_ALIAS command in its definition. In such cases it is necessary to name the parent class so that the automatically generated constructor will call the parent class constructor correctly. The simple C++ example,

```
class MySpecialClass : public AdtArrays
{
  /* AD_ALIAS MC=D_MySpecialClass */
  .
  .
  .
}
```

defines *MC* as the *Alias Name* of the class *D_MySpecialClass*. As it doesn't derive from a preprocessed class we do not use the *Parent_Class_Name* parameter. In Pascal code this would look like,

```
MySpecialClass = class(Tuseful_obj)
  {AD_ALIAS MC=D_MySpecialClass}
  .
  .
  .
end;
```

Note that *D_Class_Name* refers to the class name for the derived output class from any AD operations and applies equally to null differentiation steps. The reason the derived class is used is so that ADT can then automatically generate interface code to methods produced as a result of AD operations. The *D_Class_Name* entry must match that of the derived class as specified in the make file, which in the above case would be something like,

```
CLASS D_MySpecialClass (MySpecialClass) ...
```

3.3.2.3 AUTOINIT command

Class attributes can be automatically initialised through the constructor by placing an AUTOINIT command before their declaration in the class definition.

The command applies to all attribute declarations in the class appearing after the command. If no command is provided it defaults to MANUAL, meaning the code author is responsible for initialising the code. All attributes that are auto-initialised have corresponding initialisation arguments in the constructor for the class.

The general form of the command is,

```
AUTOINIT [phase_number]
```

where *phase_number* is an optional parameter that sets the phase of initialisation for the attributes in question. If not specified the *phase_number* defaults to 1. For example,

```
class MySpecialClass : public AdtArrays
{
  /* AD_ALIAS MC=MySpecialClass */
  /* AUTOINIT */
  int N
  ARRAY_1D M/*N*/
  /* AUTOINIT 2 */
  double P
  double Q
  .
  .
  .
  MySpecialClass(
  #include "MC_constructor_args.hpp"
  );
  .
  .
  .
}
```

or in Pascal,

```
MySpecialClass = class(AdtArrays)
  { AD_ALIAS MC=MySpecialClass }
  { AUTOINIT }
  N      : longint;
  M {N}  : ARRAY_1D;
  { AUTOINIT 2 }
  P  : double;
  Q  : double;
  .
  .
  .
  MySpecialClass({$I MC_constructor_args.pas});
  .
  .
  .
end;
```

3.3.2.4 AUTODEC command

Class attributes can be automatically null initialised through the constructor by placing an AUTODEC command before their declaration in the class definition. The command applies to all attribute declarations in the class appearing after the command. If no command is provided it defaults to MANUAL, meaning the code author is responsible for initialising the code.

The general form of the command is,

```
AUTODEC [phase_number] [NO_INTERFACE]
```

where *phase_number* is an optional parameter that sets the phase of initialisation for the attributes in question and *NO_INTERFACE* is an optional parameter that instructs ADT to not generate an R interface for the attributes to follow. If not specified the *phase_number* defaults to 1 and interfaces are created. The latter option is needed so that array types such as *ARRAY_?B* or *ARRAY_?UI* may be used within a class without resulting in errors as these and other types do not have equivalents in an R environment and therefore cannot be mapped. For example,

```
class MySpecialClass : public AdtArrays
{
  /* AD_ALIAS MC=MySpecialClass */
  /* AUTOINIT */
  int N
  ARRAY_1D M/*N*/
  /* AUTOINIT 2 */
  double P
  double Q
  /* AUTODEC */
  int R
  /* AUTODEC 2 */
  ARRAY_2D S
  .
  .
  .
  MySpecialClass(
  #include “MC_constructor_args.hpp”
  );
  .
  .
```

```
    .
  }
```

will put null-initialisation code for R into the include file *MC_ constructor_ scalars_ 1.hpp* and null-initialisation code for S into *MC_ constructor_ arrays_ 2.hpp.* For Pascal context the include file names are the same but with a *.pas* file extension. The equivalent in Pascal is,

```
  MySpecialClass = class(AdtArrays)
    { AD_ALIAS MC=MySpecialClass }
    { AUTOINIT }
    N      : longint;
    M {N}  : ARRAY_1D;
    { AUTOINIT 2 }
    P  : double;
    Q  : double;
    { AUTODEC }
    R  : longint;
    { AUTODEC 2 }
    S  : ARRAY_2D;
    .
    .
    .
    MySpecialClass({$I MC_constructor_args.pas});
    .
    .
    .
  end;
```

3.3.2.5 MANUAL command

Class attributes that are declared after a MANUAL command must be manually initialised in the author code. The general form of this command is,

```
  MANUAL
```

If no AUTOINIT, AUTODEC or MANUAL commands appear any defined attributes are treated as MANUAL defined attributes. Furthermore, manual attributes will not be accessible from R as they have no automatically generated interface code.

3.3.2.6 BLACKBOX and D/D commands

Sometimes TAPENADE may need to create differentiated code for functions that can be easily differentiated by hand. For such functions if we provide a black box definition that specifies the derivative then TAPENADE can make more efficient code by not generating differentiated versions of the function but rather using the black box definition to produce the differentiated code. We can provide black box definitions to those sorts of functions through a special comment block that appears immediately before the function declaration or implementation, but not both. For example,

```
// < D/D(x) (4 * x) - 5; >
double polyX(double x);
```

declares that the C++ function `polyX()` has the differential with respect to x of $4x - 5$. Similarly in Pascal we would have,

```
{ < D/D(x) (4 * x) - 5; > }
function polyX(x : double) : double;
```

The “less than” and “greater than” braces are required in inline definitions and are used to isolate the black box definition from other general code comments. Quite often differentials of functions with more than one variable will have a common term to the partial differentials and then specific terms for each dependent variable. This is supported in the black box specification language as is illustrated by the following contrived example.

```
// < D/D(.) -(x^2 + 4*y)^-2,
//   D/D(x) .*2*x,
//   D/D(y) .*4; >
double polyXY(double x, double y);
```

Note that ^ is used as a “raised to the power of” operator which can also be specified by the Fortran styled equivalent **. This example defines the common term as $-\left(x^2+4y\right)^{-2}$ and the extra multiplicative terms with respect to x and y respectively of $2x$ and 4, giving the complete partial differentials for `polyXY()` as,

$$\frac{\partial polyXY}{\partial x} = -2x\left(x^2+4y\right)^{-2}$$

and,

$$\frac{\partial polyXY}{\partial y} = -4\left(x^2+4y\right)^{-2}$$

The expressions defining the derivation can include function calls provided the named functions have themselves blackbox definitions or are part of the standard blackbox definitions provided by TAPENADE (Fortran instrinsics such as COS, SIN, LOG, EXP and so on). For example,

```
// < D/D(x) -sin(x) + 2 * cos(2*x); >
double trigExpression(double x);
```

The black box definitions can be defined using either single line or multiple line comments in Pascal and C++ languages. If you wish to include a "black box" comment you can make use of the # symbol to designate a single line black box comment. There is no multiline comment support for black box definitions. For example, we might comment the above with,

```
// < D/D(x) -sin(x) + 2 * cos(2*x); # derivative of trigExpression >
double trigExpression(double x);
```

In addition to specifying the nature of the differential we can also tell TAPENADE more detail about the function parameters and whether they are inputs or outputs, both or neither. This is done with a BLACKBOX statement along the lines of,

```
// < BLACKBOX ReadNotWritten:(1,1,0)
//            NotReadThenWritten:(0,0,1)
//            deps:(1,1,0,1,1,0,id);
//   D/D(.) -(x^2 + 4*y)^-2,
//   D/D(x) .*2*x,
//   D/D(y) .*4; >
double polyXY(double x, double y);
```

The keywords `ReadNotWritten`, `NotReadThenWritten`, `ReadThenWritten`, `NotReadNotWritten` and `deps` have the same meaning as is documented in the TAPENADE FAQ covering black box routines so for a detailed explaination please refer to it (see `https://www-sop.inria.fr/tropics/tapenade/faq.html#Libs1` assuming it hasn't changed addresses). For general use the BLACKBOX statement is not necessary and ADT will provide reasonable defaults for those parameters through inspection of the function prototype. Note that any function provided with an inline blackbox definition will not be included in the normal AD processing of ADT as if it were excluded using conditional compilation statements (`ifndef AD`).

If using an external library to provide some numerical calculations for which access to the source code is not provided, black box definitions will be necessary to allow TAPENADE to produce differentials for any code that makes use of it. In this case, as there is no source code, inline blackbox definitions cannot be provided, but ADT provides a alternative means of providing them. To do so we create a standalone black box definition file which is referenced in the BLACKBOX part (see 3.2.3.8) of the ADT make file. The syntax is the same as for inline definitions but sandwhiched within a Pascal styled function or procedure definition with comment delimiters removed. The "less than" and "greater than" bracing is not required in standalone definitions. For example,

```
# Black box definition for polyX
function polyX(x : real in) : real
begin
  D/D(x) (4 * x) - 5;
```

```
end
```

Alternatively, if `polyX` were implemented as a procedure we define it as,

```
# Black box definition for polyX
procedure polyX(x : real in, result : real)
begin
  D/D(x) (4 * x) - 5;
end
```

You can have as many definitions as needed within a file, or you can group them into different files and name each file in the BLACKBOX part of the ADT makefile.

For a complete definition of the black box language refer to 3.3.4.

3.3.3 Automatically Generated Files

3.3.3.1 {Alias_Name}_constructor_args file

The *{Alias_Name}_constructor_args* file contains the constructor argument list and includes all the variables necessary to initialise all the class attributes declared in AUTOINIT scope. The file should be included in the declaration and implementation of the constructor. For example,

```
class MySpecialClassB : public MySpecialClass
{
  /* AD_ALIAS MCB=D_MySpecialClassB, MySpecialClass */
  .
  .
  .
  #include "MCB_array_plans.hpp"

  MySpecialClassB(
  #include “MCB_constructor_args.hpp”
  );
  .
  .
  .
};
```

for the declaration and,

```
MySpecialClassB::MySpecialClassB(
  #include “MCB_constructor_args.hpp”
  )
 : MySpecialClass(
     #include "MCB_constructor_call_args.hpp"
```

```
    )
  {
    #include "MCB_constructor_locals.hpp"
    #include "MCB_constructor_scalars_phase_1.hpp"
    #include "MCB_constructor_arrays_phase_1.hpp"
    #include "MCB_array_plans_init.hpp"
    .
    .
    .
  };
```

for the implementation in C++. For Pascal the example would be,

```
  MySpecialClassB = class(MySpecialClass)
    {AD_ALIAS MCB=MySpecialClassB, MySpecialClass}
    .
    .
    .
    {$I MCB_array_plans.pas}

    constructor create({$I MCB_constructor_args.pas});
    .
    .
    .
  }
```

and,

```
  constructor MySpecialClassB.create({$I MCB_constructor_args.pas});
  var
    {$I MCB_constructor_locals.pas}
  begin
    inherited create({$I MCB_constructor_call_args.pas});
    {$I MCB_constructor_scalars_phase_1.pas}
    {$I MCB_constructor_arrays_phase_1.pas}
    {$I MCB_array_plans_init.pas}
    .
    .
    .
  end;
```

3.3.3.2 {Alias_Name}_constructor_call_args file

The *{Alias_Name}_constructor_call_args* file contains the argument list required to call the parent constructor. This file should be included in the implementation of the constructor. See section 3.3.3.1 for an example of its usage.

3.3.3.3 {Alias_Name}_constructor_locals file

The *{Alias_Name}_constructor_locals* file contains declarations of local variables needed to carry out the automatic initialisation of class attributes in the constructor. This file should be included in the constructor implementation. See section 3.3.3.1 for an example of its usage.

3.3.3.4 {Alias_Name}_constructor_scalars_phase file

The *{Alias_Name}_constructor_scalars_phase* file contains code to intialise scalars. There are as many phase files as there are phases defined through AUTOINIT and AUTODEC commands. See section 3.3.2.3 and 3.3.2.4 for more details on initialisation phases.

The author of the code is responsible for including the *{Alias_Name}_constructor_scalars_phase* files in the correct order and location and is free to add any custom initialisation code in between include commands. See section 3.3.3.1 for an example of usage.

3.3.3.5 {Alias_Name}_constructor_arrays_phase file

The *{Alias_Name}_constructor_arrays_phase* file contains code to intialise arrays. There are as many phase files as there are phases defined through AUTOINIT and AUTODEC commands. See section 3.3.2.3 and 3.3.2.4 for more details on initialisation phases.

The author of the code is responsible for including the *{Alias_Name}_constructor_arrays_phase* files in the correct order and location and is free to add any custom initialisation code in between include commands. See section 3.3.3.1 for an example of usage.

3.3.3.6 {Alias_Name}_impl_lib_interface_methods file

The *{Alias_Name}_impl_lib_interface_methods* file contains automatically generated code to interface your class with R. This file is automatically included in the derived class implementation generated by ADT.

3.3.3.7 {Alias_Name}_impl_lib_interface_globals file

The *{Alias_Name}_impl_lib_interface_globals* file contains automatically generated code to interface your class with R. This file is automatically included in the derived class implementation generated by ADT.

3.3.3.8 {Alias_Name}_decl_lib_interface_methods file

The *{Alias_Name}_decl_lib_interface_methods* file contains the automatically generated interface method declarations that interface your class with R.

This file is automatically included in the derived class implementation generated by ADT.

3.3.3.9 {Alias_Name}_decl_lib_interface_globals file

The *{Alias_Name}_decl_lib_interface_globals* file contains the automatically generated interface function declarations that interface your class with R. This file is automatically included in the derived class implementation generated by ADT.

3.3.3.10 {Alias_Name}_decl_lib_interface_constructor file

The *{Alias_Name}_decl_lib_interface_constructor* file contains the automatically generated constructor and destructor function declarations that interface your class with R. This file is automatically included in the derived class implementation generated by ADT.

3.3.3.11 {Alias_Name}_impl_lib_interface_constructor file

The *{Alias_Name}_impl_lib_interface_constructor* file contains automatically generated constructor and destructor function implementations that interface your class with R. This file is automatically included in the derived class implementation generated by ADT.

3.3.3.12 {Alias_Name}_impl_lib_registration file

The *{Alias_Name}_impl_lib_registration* file contains automatically generated R interface registration code which must be part of the R library being compiled. This file should be included in the project file for a Pascal library, for example:

```
library RosenbrockPas;
uses
  Classes, MinRosenbrock_Unit,
  D_MinRosenbrock_Unit, R_MinRosenbrock_Unit,
  DR_MinRosenbrock_Unit, adtarray, Raccess;
// -------------------------------------
{$I RRb_impl_lib_registration.pas}
exports
  {$I RRb_lib_exports.pas};
begin
end.
```

In the case of a C++ project a source file should be added to the project that includes this file along with the header files for the derived classes being interfaced to. In the Rosenbrock project case the added file is *registration.cpp* and contains the following:

```
// ------------------------------------
// Code to register R interface for the Test minimisation example code for ADT.
// ------------------------------------
#include "DR_Rosenbrock.hpp"
#include "RRb_impl_lib_registration.hpp"
```

3.3.3.13 {Alias_Name}_lib_exports file (Pascal only)

The *{Alias_Name}_lib_exports* file contains a comma seperated list of all the functions to export as the dynamic library interface. This file is required for Pascal projects and should be included in the *exports* statement of your project file. For example,

```
exports
  .
  .
  .
  {$I MCB_lib_exports.pas};
```

C++ projects do not use this file and export symbols through the function declaration itself.

3.3.3.14 {Alias_Name}_array_plans file

The *{Alias_Name}_array_plans* file contains the declaration of necessary *AdtArrayPlan* objects needed to initialise arrays in *{Alias_Name}_constructor_arrays_phase* files. This file should be included along side the other attributes of your class. See section 3.3.3.1 for a examples of usage.

3.3.3.15 {Alias_Name}_R_interface.r file

The *{Alias_Name}_R_interface.r* file contains *R* code implementing get and set method accessors and code to instantiate and interact with your class.

The R *create* function is used to create an instance of the class and comforms to the following naming convention.

```
{Alias_Name}.create <- function({initialisation argument list})
```

The initialisation argumenst contain one argument per AUTOINIT class attribute in the constructor call. For example, the Rosenbrock demonstration project has the *create* function,

```
RRb.create <- function(N)
```

The *create* function returns a pointer to the object instance to R which must be saved for use as the instance pointer in calling the object instance methods. For example,

```
RRb.Context <- RRb.create(150)
```

Before calling the *create* function it is necessary to load the dynamic library using a call to *dyn.load()*. For example,

```
dyn.load("../../objs/sample/Rosenbrock/.libs/librosenbrock.so")
```

For each attribute that is AUTOINITialised, ADT creates *get* and *set* R wrapper functions to easily access your attribute data. Furthermore, these accessors assume the identical array structure as in the class, including the indexing base. This is acheived through the use of the *Oarray* package to provide abstracted array indexing. The motivation behind this one to one correspondence is to aid in debugging the library code. The *get* and *set* methods are named through the templates,

```
{Alias_Name}.get.{Attribute_Name} <- function(Context, ...)
```

and,

```
{Alias_Name}.set.{Attribute_Name} <- function(Context, Arg, ...)
```

Notice that both the set and get functions can index a subset of the attribute being called. For example, if we hade a 3 dimensional array called *PopAgeAreaTime* then we could inspected the entire array content with,

```
PopMod.get.PopAgeAreaTime(Context)
```

the two dimensional array subset at Age 2 with,

```
PopMod.get.PopAgeAreaTime(Context, 2)
```

the one diemsional array subset at Age 2 and Area 3 with,

```
PopMod.get.PopAgeAreaTime(Context, 2, 3)
```

and the single value at Age 2, Area 3 and Time 1 with,

```
PopMod.get.PopAgeAreaTime(Context, 2, 3, 1)
```

The same equally applies to the setter methods though in that case the second argument in the call must correspond to the value/values you are setting it to and needs to be of size and shape suitable to initialise the attribute in question. For example,

```
PopMod.set.PopAgeAreaTime(Context, ScalarValue, 2, 3, 1)
```

For each public method in the class ADT creates an R wrapper call with the name based on the template,

```
{Alias_Name}.{Method_Name} <- function(Context, {function argument list})
```

and

```
{Alias_Name}.nt.{Method_Name} <- function(Context, {function argument list})
```

The R wrapper function has one plus the same number of arguments as the method it wraps, the first argument always being the object instance pointer. There are two variants of the interface methods that are created for each class method, both with the same call arguments. The reason for this has to do with format translation. To ellaborate, arrays in R are represented internally in column major format whereas C++ and Pascal languages represent arrays in row major format. To aid in debugging code it is nice to have a common representational framework whether in a R context or a C++ / Pascal context so ADT normally translates the array representation when passing parameters from R code to C++ / Pascal code and visa versa via a transpose operation. This is true for all constructor arguments and the *get* and *set* methods. However, if a function must be called repeatedly (for example, when passing your class function to the *nlminb()* minimizer function in R) it can be wasteful of computing resources to be continually transposing matrices to make such method calls. Therefore we have two implementations : one with transposition and one without. The one without is given the *.nt.* modified name, *.nt.* meaning *no transpose*.

3.3.4 Black Box Grammar

The lexer ignores C++ and Pascal comment delimiters. Inline blackbox definitions must be braced by "less than" (<) and "greater than" (>) symbols. Line comments can be included in black box definitions by begining the line with the # character. Note that in an inline definition the comment start must appear after the language comment delimiter. For example,

```
// # This is a valid black box comment in C++ code
# // This is NOT a valid black box comment in C++ code
{ # This is a valid black box comment in Pascal code }
# { This is NOT a valid black box comment in Pascal code }
```

```
Keywords
   begin BLACKBOX boolean boolean_array character character_array
   complex complex_array D/D end function id in integer integer_array
   NotReadNotWritten NotReadThenWritten out procedure ReadNotWritten
   ReadThenWritten real real_array result
```

Tokens

```
Identifier:
   An identifier starting with a non-numeric character and containing
   only numbers, letters and underscore characters.

DecimalLiteral:
   An integer number.

FloatingPointLiteral:
   A floating point number.
```

Rules

```
root:
   derivativeList

derivativeList:
   derivative
   derivativeList derivative

MakeOptionsList:
   MakeOption
   MakeOptionsList MakeOption

derivative:
   'D/D' '(.)'  exprAdditive ',' derivativeExtendedList ';'
   'D/D' '(' Identifier ')' exprAdditive ';'
   'BLACKBOX' blackBoxSpecList ';'
   'procedure' Identifier '(' argDefList ') 'begin' derivativeList
   'end'
   'function' Identifier '(' argDefList ')' ':'  argType 'begin' derivativeList
   'end'

derivativeExtended:
   'D/D' '(' Identifier ')' '.'  '*' exprAdditive

derivativeExtendedList:
   derivativeExtended
   derivativeExtendedList ',' derivativeExtended

argType:
   'real'
   'integer'
   'complex'
   'character'
   'boolean'
   'real_array'
   'integer_array'
   'complex_array'
   'character_array'
   'boolean_array'

dirType:
   'in'
   'out'
   'in' 'out'
   'out' 'in'

argDef:
   Identifier ':'  argType dirType
   Identifier ':'  dirType argType
   'result' ':'  argType

argDefList:
   argDef
   argDefList ',' argDef

blackBoxSpecList:
   blackBoxSpec
   blackBoxSpecList blackBoxSpec

blackBoxSpec:
   'ReadNotWritten' ':'  '(' identList ')'
   'NotReadThenWritten' ':'  '(' identList ')'
   'NotReadNotWritten' ':'  '(' identList ')'
   'ReadThenWritten' ':'  '(' identList ')'
   'deps' ':'  '(' identList ')'

ident:
   'id'
   DecimalLiteral

identList:
   ident
   identList ',' ident

exprAdditive:
   exprMultiplicative
   exprAdditive '+' exprMultiplicative
   exprAdditive '-' exprMultiplicative

exprMultiplicative:
   exprPower
   exprMultiplicative '*' exprPower
   exprMultiplicative '/' exprPower

exprPower:
   exprUnary
   exprPower '^' exprUnary
   exprPower '**' exprUnary

exprUnary:
   exprPostfix
   '+' exprPostfix
   '-' exprPostfix

exprPostfix:
   DecimalLiteral
   FloatingPointLiteral
   Identifier
   Identifier '(' argList ')'
   Identifier '(' exprAdditive ')'

argList:
   exprAdditive
   argList ',' exprAdditive
```

Chapter 4

ADT Internals

Here we outline the internal structure and operation of ADT for the specific purpose of aiding in the maintenance of the software. The source code behind the software is large and is daunting to contemplate without first hand knowledge, but the software has a coherent structure should aid in any future development. In order to maximise portablility and minimise dependencies, ADT does not make use of anything other than ANSI C++ and STL. Anything we need we construct from this foundation and deliberately avoid using large monolithic libraries such as boost. This keeps ADT a compact package with minimal build requirements. Any future development should adhere to this practice.

4.1 Common Code

A significant portion of the code in ADT is common code that is used throughout in many contexts. Here we detail the majority of that common code.

4.1.1 String Class

The string class used extensively in ADT is typedefed as *string* ind is a template instantiation of the template class *string_ext* which derives from the STL string class *std::basic_string*. The derived and extended version adds string concatenation operations using overloaded + operators, a c string pointer cast operator, some common string manipulation methods, caseless string comparison, and most importantly, overrides the constructors and assignment operators to fix a dangerous behaviour in the STL version. The class is declared and implemented in *adtcommon.hpp*

In particular, the STL *basic_string* class has an optimisation whereby if you assign to a string object another string object it will share pointers to the internal string representation to conserve memory. Whilst it sounds like a nice idea, in practice it causes no end of trouble because you may happen to be assigning from a temporary string object (say an object on the stack or an aggregate of a class that will soon be destroyed) which then becomes invalid when the parent dies. If it were implemented properly with a reference counted pointer to the internal buffer then such a strategy would be fine, but the naive implementation within does not do so, thus making it more trouble than it is worth. Therefore, all the ways of assigning a string object to a string object (constructors and assignment operators) are overridden and call the private method *forceCopy()* to ensure that pointers are never shared.

4.1.2 ReferenceCount Class

In ADT many objects are shared objects to be shared between many clients. Those classes that are to be shared need to be reference counted to manage the life cycle. This is handled by deriving all shared classes from the *UtlReferenceCount* class defined in *adtcommon.hpp*. It has three methods : *lock()* to increment the reference count, *unlock()* to decrement the reference count and *lockCount()* to query the lock count. When calling the *unlock()* method you should check the return value and if *true* the *delete* operator should be called on the object instance to free the object. This is handled cleanly by using the helper macro *UtlReleaseReference()* to release an object no longer needed.

This behaviour could have been built into the *unlock()* method but doing so then makes automatic instances of the class impossible (ie. creating a class instance without using *new()*). It is left as the client responsibility to free the class so that stack/automatic instances of the class can be used.

4.1.3 UtlFilePath Class

File names are commonly manipulated in ADT so to simplify this processing and to share the code needed for the manipulation of file names we have created the *UtlFilePath* class. This class takes a filename with or without full path extension and splits it up into *Drive*, *Directory*, *Name* and *Extension* components, which can subsequently be modified and joined to form a new file name and path. This class correctly handles the differences of file name representations in Linux / Unix and Windows / DOS. Provided you use this class to manipulate file names then you need not worry about the file name represenational differences between different operating systems. This class is declared in *adtcommon.hpp* and implemented in *adtcommon.cpp*.

4.1.4 AdtParse Namespace

Though most of the parsing required in ADT is handled through Flex and Bison generated lexers and parsers, some internal stuff needs to be handled manually (parsing automation commands and array size specifications within comment blocks for example). To that end we have a series of functions that aid in writing parsers declared in the *AdtParse* namespace in the file *adtutils.hpp*. The code is largely self explanatory and usage can be easily determine by grep'ing for usage within ADT so I'll refrain from going into any detail about this, suffice to say, that if you need to do a hand written parser then consider using the functions already available in *AdtParse* rather than introducing extra code.

4.1.5 AdtFile Class

Within ADT we spend a lot of time creating formatted source files in different languages. To aid in creating nicely formated output files we make use of the *AdtFile* class. This class has methods to write strings and chars, insert new lines and tabs and change indentation (the level of tabbing to be applied after a new line character). The output can be directed either to a file or a string object. If *AdtFile* is opened as a Fortran file, it also correctly handles the creation of Fortran continuation lines.

If you need to write formated output to a text file you should make use of *AdtFile* as it will make the code preparation considerably easier. This class is declared in *adtutils.hpp* and again, usage is pretty self explanatory with many examples easily found by grep'ing for usage within ADT.

4.1.6 AdtFileCopy Class

In various places we are required to create copies of text files. As an aid to this we have created a file copying class called *AdtFileCopy* which is declared in *adtutils.hpp*. Search for it in *adtmake.cpp* for an example of it's usage.

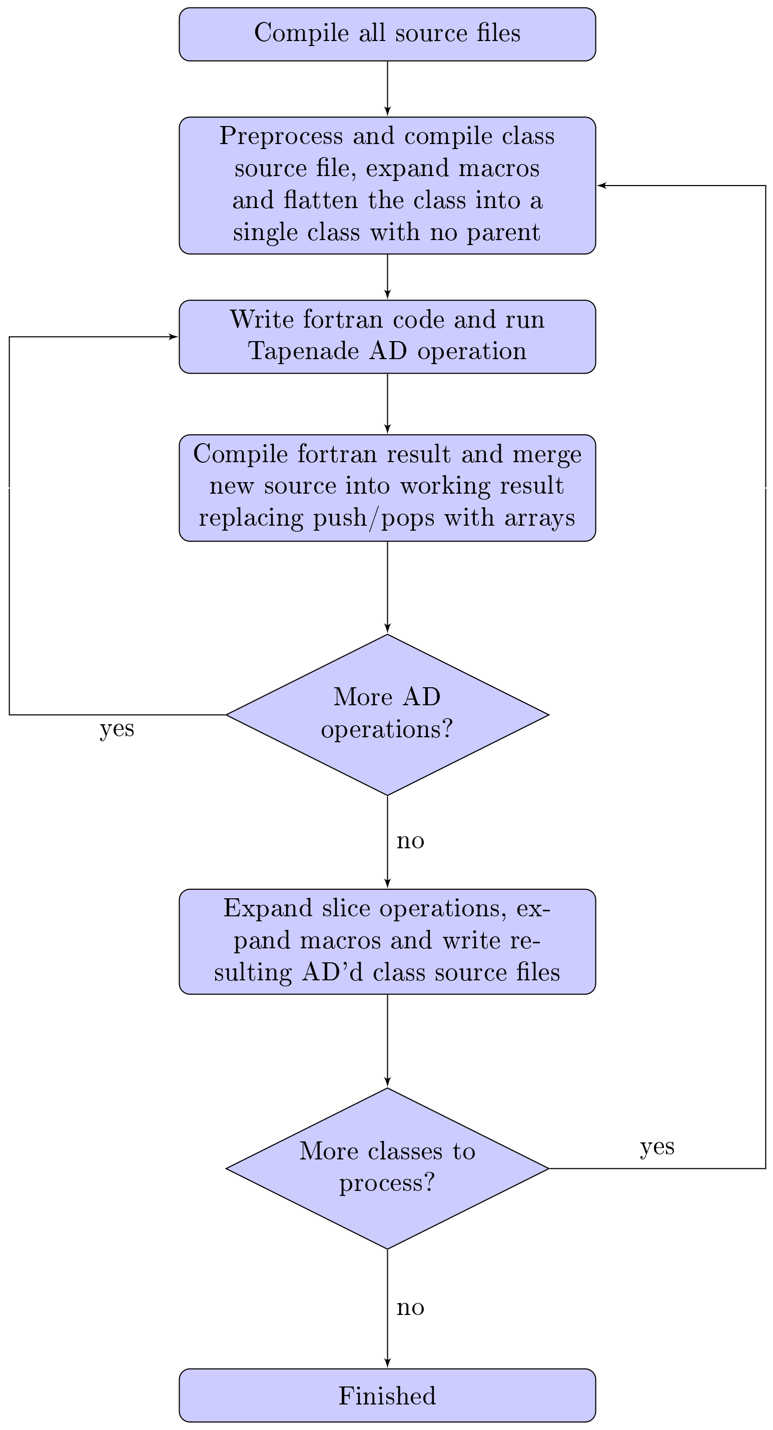
4.1.7 AdtStringCache Class

When parsing the Pascal, C++ and Fortran source files, ADT creates complete object based parse trees for the code. In the token stream there is a lot of text that needs to be allocated in strings but much of that text is repeated text representing the same token so to store each individually wastes a lot of memory and hinders the performance of ADT. To avoid this problem all string tokens in the lexer stream are allocated through a global instance of the *AdtStringCache* class, which returns a pointer to a single allocation of the one name through the *add()* method. Thus, if we add the string *MyVar* three times in a row it will only allocate one copy in memory and return the same representation to the caller in the other two cases. This class is declared in *adtutils.hpp* and is used by all the flex generated lexers in ADT.

4.2 Make System

The make system is the over-arching component of ADT responsible for orchestrating the correct sequence of operations needed to build a particular ADT make file. The makefile parsing is carried out by a flex / bison generated parser. The tokeniser is implemented in *make_l.l* and the parser in *make_y.y* . The make system code resides in *adtmake.hpp* and *adtmake.cpp*.

4.2.1 Processing Flow Chart


Compile all source files
Preprocess and compile class source file, expand macros and flatten the class into a single class with no parent
Write fortran code and run Tapenade AD operation
Compile fortran result and merge new source into working result replacing push/pops with arrays
More AD operations?
yes
no
Expand slice operations, expand macros and write resulting AD'd class source files
More classes to process?
yes
no
Finished

The simplified make processing is as shown in the flow chart. It begins with all the source files named in the SOURCE FILES section of the make file being compiled.

Then, for all classes we start by pre-processing and compiling the named source file and expanding macros. The pre-processing referred to here is those comment embedded commands used to define and implement an R interface to the class (see Section 3.3). The macro expansions are those defined in CPP OPTIONS FILE or PASCAL OPTIONS FILE (depending on whether the source code is C++ or Pascal).

The parse tree of the flattened class source is then written as equivalent Fortran 95 code and TAPENADE is invoked, carrying out the desired AD operation on the Fortran code.

The resulting AD'ed Fortran code is then compiled and any new Functions or Subroutines that TAPENADE created are integrated into the working Fortran parse tree. Within that process of integration, all instances of *PUSH*, *POP*, *PUSHARRAY* and *POPARRAY* must be substituted with stack arrays. This step is a requirement to allow TAPENADE to apply further AD operations on top of reverse mode AD'ed code.

We continue with this processing until all AD operations defined for the given CLASS have been carried out. When all have been carried out we finalise the processing for that class by first expanding slice operations into an equivalent set of calls to *XCOPY* and / or *ZERO*, then translating the working Fortran source into an equivalent C++ or Pascal source (depending on the source code language). We do this for all CLASS definitions in the makefile.

The bulk of the processing handled by the make system is implemented in the *AdtMakeSystem::make()*, *AdtMakeCommand::make()* and *AdtMakeClass::make()* methods.

4.2.2 Incremental Compile

The implicit assumption on which the ADT incremental compile system is based is that every successive step in an ADT make file is dependent upon the previous one. That being the case, when an AD operation is changed in the middle of a list of AD operations (contained in a CLASS entry within the makefile) then that AD operation and every other operation following is re-evaluated. when ADT is re-run.

This mechanism is managed by the *AdtMakeIncremental* class defined in *adt-make.hpp* and implemented in *adtmake.cpp*. The principle of operation is through simple string comparison. An instance of *AdtMakeIncremental* is created and then opens the dependency check file through the *open()* method. If the file exists (which is the case when an ADT make file has previously been processed) then it reads the entire contents of the file into the *ReferenceString* attribute of the class. Then, scattered throughout the make processing are calls to the *checkText()*, *checkList()*, *checkMap()* and *checkFile()* methods. Each of these methods serialises the objects being checked in a string format and stores it in the *CheckString* attribute. This string is then checked against *ReferenceString*. If it matches then we know nothing has changed so no re-building is required but if it doesn't then we flag it as needing to be rebuilt. When the entire make has been run *CheckString* is then written to the dependency check file, updating the dependency information ready for any subsequent build.

One other thing the make system needs from the increment compile infrastructure is knowledge about new methods and attributes resulting from the carried out AD operations. Normally this information is garnered through carrying out the AD operations but if the AD operation is bypassed because it was carried out in a previous make then we need to restore this information. This is the purpose of the *updateNewMethodsAndAttributes()* method, which both extracts previously stored new method and attribute lists from the dependency check file and merges new changes resulting from any carried out operations into these lists to create an update set that is written to the dependency check file at the completion of the make operation.

4.3 Code Parsers

4.3.1 Common Code Parser Infrastructure

The code (as in Pascal, C++ and Fortran) parsers share much code in common. Many of the operations required to manage the parse results are the same so it makes sense to keep as much of this shared functionality in common code.

All the parsers in ADT function in the same way, by parsing the source file into an object based parse tree with objects corresponding to the non-terminals in the grammar file definition for the language, giving a one to one correspondence between the grammar and the parse tree output. This approach, whilst not a memory efficient way of handling it, provides benefits in other ways by making the translation and manipulation of the code simple and straight forward.

The evaluated rules are translated into an object tree by creating the objects through C wrapper functions that are called in the C statements for the rule. For example, the non-terminal for *conditional_ expression* in the C++ grammar is,

```
conditional_expression : logical_or_expression
{
  $$.pContext = adtCppConditionalExpression_create($1.pContext, 0, 0);
}
  | logical_or_expression QUESTION expression COLON assignment_expression
{
  $$.pContext = adtCppConditionalExpression_create($1.pContext,
                                                   $3.pContext,
                                                   $5.pContext);
};
```

The root of the tree at any point in time is in the *$$.pContext* state for the parser. The parser state is represented by the corresponding language struct in the lexer.h file. For C++ it is,

```
typedef struct cppType
{
  void*       pContext;
  const char* sValue;
  double      dValue;
  int         nBlockDepth;
  const char* sComment;
} cppType;
```

The state is of the same structure for the other supported languages. As mentioned before, *pContext* holds the root of the parse tree to that point in processing. *sValue* and *dValue* hold string and numerical results that require persisting in the parser state, *nBlockDepth* is used to track the code scoping level (how many nested scopes we are in at this point in parsing), and *sComment* holds any comment string associated with the parsing of the current rule.

In the case of the above example, a logical expression object is created and assigned to *pContext* through the call the the corresponding non-terminal create function, in this case *adtCppConditionalExpression_create()*. The create function has as many parameters as required to fully describe all the rules that lead to the non-terminal in question. In the cases where extra parameters aren't required the unused inputs are set to appropriate defaults (typically 0).

Note that the $1, $2, ... variables return the state corresponding to the corresponding non-terminal in the rule. In the above example *$1.pContext* returns the *logical_or_expression* object, *$3.pContext* returns the *expression* object and *$5.pContext* returns the *assignment_expression* object.

For list non-terminals the requirements is a little bit different. For example, again in the C++ grammar we have,

```
simple_declaration_list : simple_declaration
{
  $$.pContext = adtCppSimpleDeclarationList_create($1.pContext);
}
  | simple_declaration_list simple_declaration
{
  $$.pContext = adtCppList_add($1.pContext, $2.pContext);
};
```

In the case of the first rule we again need to call the corresponding create function to create the list, but for list insertions we then call the *adtCppListAdd()*

function[1] to insert the object into the list.

The C wrapper functions are again all very similar in construction. For example,

```
void* adtCppConditionalExpression_create(void* pLogicalOrExpressionObj,
                                         void* pExpressionObj,
                                         void* pAssignmentExpressionObj)
{
 return (Hnd(new AdtCppConditionalExpression(
                      ObjAndRelease(pLogicalOrExpressionObj),
                      ObjAndRelease(pExpressionObj),
                      ObjAndRelease(pAssignmentExpressionObj)),
             yyCpp_lineNumber(),
             yyCpp_fileName()));
}
```

Here we coerce the object parameters into the correct type (*AdtParser**), pass them as parameters to the constructor of the object that needs creating and release the object (decrement a reference count) after the conditional expression object has been created. This happens through the *ObjAndRelease()* macro call. All objects in the parse tree are derived from the *AdtParser* class which implements reference counting for object life cycle management. It is for this reason that we need to release the object after the requested object is create or else memory leaks will result.

The class *Hnd* is a conversion class to simply convert the *AdtParser** back into a *void** needed by the parser and to assign the contextual information (line number and filename from *yyCpp_lineNumber()* and *yyCpp_fileName()*) to the object instance. This contextual information can then be used to give more informative error messages in the case of code that does not parse without error.

Comment information, which is usually discarded in normal parsers, in our case holds important information that is needed for interpretting how to process the code (embedded commands) and as such, is preserved in the parse tree. The general approach is two fold. Firstly, all identifier objects that have inline comments trailing them have the comment tokenised in place with the identifier token, which is later seperated out into identifier and comment within the class representing the identifier. Secondly, all other comments are bound to the keyword token that appears immediately after the comment. If more than one comment construct appears before the keyword token then they are concatenated together and bound to the associated keyword object, whichever that may be. The binding takes the form of a comment string attribute defined in *AdtParser* class accessible via the *AdtParser::comment()* method.

This basic approach is repeated for all non-terminals in all the supported lan-

[1]the same applies to other languages but the function name changes, for example, we use *adtDelphiList_add()* for Pascal code

guages.

Each class representing a non-terminal is derived (indirectly) from *AdtParser*. In the case of C++ it typically derives directly from *AdtCppBase()* which in turn derives from *AdtParser*, with a similar but appropriately named class for the other languages. *AdtParser* is an abstract base class for which we need to implement a range of methods for attribute object management and run time type identification. To simplify this repetative coding there are a number of macros used for this very purpose. We require the *declType* macro to appear in the public section of the class definition and the *implType* macro to appear in the implementation of the class. We also need to declare and implement code translation methods to write the parse tree in the formats we need to support, in this instance this means the *writeCPP* and *writeFortran* methods. For example,

```
class AdtCppConditionalExpression : public AdtCppBase
{
private:
  AdtCppLogicalOrExpression*    LogicalOrExpression;
  AdtCppExpression*             Expression;
  AdtCppAssignmentExpression*   AssignmentExpression;

public:
  AdtCppConditionalExpression(AdtParser* pLogicalOrExpressionObj,
                              AdtParser* pExpressionObj,
                              AdtParser* pAssignmentExpressionObj);

  AdtCppConditionalExpression(const AdtCppConditionalExpression& rCopy);

  virtual ~AdtCppConditionalExpression();

  virtual bool isSimple() const;

  virtual AdtFile& writeCPP(AdtFile& rOutFile,
                            int nMode = 0) const;

  virtual AdtFile& writeFortran(AdtFile& rOutFile,
                                int nMode = 0) const;

  declType;
};
```

and,

```
implType(AdtCppConditionalExpression, AdtCppBase);
```

Note that the class definition has corresponding object attributes for each non-terminal named in the rule, in this case *LogicalOrExpression*, *Expression* and

AssignmentExpression. Initialising these object attributes requires a lot of repetitive coding which is simplified through macros. This we can see in the implementation of the class constructors below.

```
AdtCppConditionalExpression::AdtCppConditionalExpression(
  AdtParser* pLogicalOrExpressionObj,
  AdtParser* pExpressionObj,
  AdtParser* pAssignmentExpressionObj)
  : AdtCppBase()
{
  initObject(LogicalOrExpression,
             pLogicalOrExpressionObj,
             AdtCppLogicalOrExpression,
             false);

  initObject(Expression,
             pExpressionObj,
             AdtCppExpression,
             false);

  initObject(AssignmentExpression,
             pAssignmentExpressionObj,
             AdtCppAssignmentExpression,
             false);
}
```

and

```
AdtCppConditionalExpression::AdtCppConditionalExpression(
  const AdtCppConditionalExpression& rCopy)
 : AdtCppBase(rCopy)
{
  copyObject(LogicalOrExpression,
             rCopy,
             AdtCppLogicalOrExpression);

  copyObject(Expression,
             rCopy,
             AdtCppExpression);

  copyObject(AssignmentExpression,
             rCopy,
             AdtCppAssignmentExpression);
}
```

We use the *initObject()* macro to initialise the object instance and the *copy-Object()* macro to copy and object instance from another class instance. These macros are defined in *adtparser.hpp*. In the destructor we also need to release the object instances we hold on to which we do so with the *UtlReleaseReference()* macro as shown below.

```
AdtCppConditionalExpression::~AdtCppConditionalExpression()
{
  UtlReleaseReference(LogicalOrExpression);
  UtlReleaseReference(Expression);
  UtlReleaseReference(AssignmentExpression);
}
```

This basic coding pattern is repeated throughout for all non-terminals in all the supported grammars.

The benefit of this relatively complex approach to building the parser comes when we need to interpret it because the base class *AdtParser* provides a great range of services to be able to interpret, search, traverse and change the parse tree with little additional coding.

4.3.1.1 Object Identity

Run time type identification is of particular importance in being able to interpret the parse tree without bloating the code base. Rather than relying on the RTTI mechanism available in C++, we implement our own string based RTTI which better integrates into the tasks we require of it.

Each class that incorporates the *declType()* and *implType()* macros will implement an overidden version of the virtual method *isType()*, which takes a single argument being a string naming the class type (which is just the name of the class in the code you think it is) and returns true if of that type or false otherwise. The implementation of *isType()* first calls the parent version of *isType()* so it will correctly identify its type through the inheritance tree.

For example, if we wanted to know if an *AdtParser* pointer object *pParserObj* was an *AdtCppConditionalExpression* object we would do something like,

```
if (pParserObj->isType(“AdtCppConditionalExpression”))
{
  // It is an AdtCppConditionalExpression object!
  .
  .
  .
}
```

Classes that represent list non-terminals should also use the *declListType()* macro to declare the type of object class the list will hold. This implements the method *listType()*, which returns a string naming the class of the list objects.

4.3.1.2 Navigating the Parse Tree

Given the pointer to a AdtParser object of known type, we can navigate to a different part of the parse tree with knowledge of the language grammar (the objects can only be assembled in a particular manner dictated by the grammar).

When a given class is instantiated we use the *initObject()* and *copyObject()* macros to initialise the class attributes. Internally, these macros register the attributes by name with the static *DescendantObjNameList* and *DescendantObjNameMap* attributes of the class. *DescendantObjNameList* holds the names of all the attributes and *DescendantObjNameMap* holds the relative position of the object in the classes memory layout keyed on the attribute name. The *DescendantObjNameMap* is used for navigation and replacment of the class attributes with new versions.

Now, if we want to find a particular object in the tree we can make use of the method,

```
    AdtParser* findDescendant(const char* pDescendantPath) const;
```

It takes as an argument, a string naming the objects in the search path from our root to the desired descendant location. The name list is comma separated. Multiple search paths can be provided by concatenating them with a semi-colon separator. The names refer to the names of the objects in the path. If you want to decend through a list object then you name the class type of the list. For example, *this* is an *AdtCppConditionalExpression* object we could descend to the *AdtCppInclusiveOrExpression* in the tree (if indeed there is one) with the code,

```
  AdtParser* pInclusiveOrExpression =
  findDescendant("LogicalOrExpression,LogicalAndExpression,InclusiveOrExpression")
```

If we want to enumerate all objects in a descendent list non-terminal we can use the method,

```
    void enumerateDescendantList(AdtParserPtrList& rList,
                                 const char* pDescendantPath) const;
```

which returns all objects in the path in *rList*. Alternatively, if we want the objects in a map keyed on the object name (the string returned by the *name()* method of the object) then we would use,

```
    void enumerateDescendantMap(AdtParserPtrByStringMap& rMap,
                                const char* pDescendantPath) const;
```

and if we know the path contains multiple instances of an object with the same name we can obtain all instances keyed on name with a multimap by using the method,

```
void enumerateDescendantMap(AdtParserPtrByStringMultiMap& rMap,
                            const char* pDescendantPath) const;
```

As an example, we could enumerate all the declared variables in an *AdtDelphiGoal* object instance with the call,

```
enumerateDescendantMap(ExportedVariableMap,
                       "Unit,InterfaceSection,VarSection,VarDeclList");
```

We can also ascend the tree of objects with,

```
AdtParser* findAscendantWithClassLineage(const char* pAscendantClassPath,
                                         AdtParser** ppSibling = 0) const;
AdtParser* findAscendantWithClass(const char* pAscendantClass,
                                  AdtParser** ppSibling = 0) const;
```

The first version searches through the objects parent list and matches the parent class types with the *pAscendantClassPath*. The second version simple looks for the first instance of the class named in *pAscendantClass* in the parent list. If the operation fails a null pointer is returned.

We can find objects that occur in the tree using the methods,

```
void findObjects(AdtParserPtrList& rList,
                 const char* pClassName,
                 const char* pObjectName = 0,
                 bool bMatchCase = false,
                 const char* pParentClassName = 0,
                 bool bAllowPartialNameMatch = false) const;
AdtParser* findObject(const char* pClassName,
                      const char* pObjectName = 0,
                      bool bMatchCase = false,
                      const char* pParentClassName = 0,
                      bool bAllowPartialNameMatch = false) const;
```

The first version finds all instances of the specified objects and returns them in *rList*, whereas the second version returns a single object instance.

4.3.1.3 Object Modification

When we wish to modify the parse tree (say for example, in replacement push and pop operations in AD Fortran code with array equivalents) we can do so with a host of different methods supporting modification of the class attributes. These include,

```
bool add(AdtParser* pObj, bool bAppend = true);
bool insertBefore(const AdtParser* pObj, AdtParser* pInsertObj);
bool insertAfter(const AdtParser* pObj, AdtParser* pInsertObj);
void remove(AdtParser* pObj);
void remove(const char* pObjName);
void removeAllExcept(const AdtParser* pObj);
void removeAllExcept(const char* pObjName);
bool replace(AdtParser* pObj, AdtParser* pListObj);
bool replaceInParent(AdtParser* pObj);
```

Many of these methods are self explanatory in there use. For further clarification you should search for the usage of these methods in the ADT source code to see how you might use them.

This is by no means a comprehensive discussion of the services available in *AdtParser* so you should study the class definition to uncover more of what is there, but it does serve as a comprehensive introduction of the services available so that you don't end up writing unecessary code.

4.3.1.4 Parser Abstraction Layers for the Make System

The make system is required to compile multiple languages (C++, Pascal and Fortran) as part of its processing. Rather than complicate the code with language based conditionals, the support for multipled languages has been abstracted out into two abstract classes, *AdtCompilerBase* and *AdtSourceRoot*.

AdtCompilerBase is an abstract base class representing the compiler for a given language. All compiler / parsers derive from this to produce a concrete implementation for the given language. An instance of the *AdtCompiler* class is used to compile source files, which internally determines the appropriate compiler class to use in the *createCompilerForFile()* method and assigns the dynamic instance to the *ParserContext* attribute of the class. Then the compiler methods,

```
void parseCommandBlock(const char* pCommandBlock, bool bEmbedded);
bool parse(const char* pFilename,
           const char* pCommandBlock = 0,
           const char* pCommandBlockName = 0,
           const AdtStringList* pSearchPaths = 0,
           bool bForwardMode = true);
bool parseString(AdtParser*& pRoot,
                 const char* pString);
void releaseRoot();
AdtParser* parseRoot() const;
AdtSourceRoot* sourceRoot() const;
```

simply delegate to the corresponding methods in *ParserContext*.

AdtSourceRoot is an abstract base class from which all root non-terminal classes in the language grammar should derive from and implement the abstract methods. These methods are the interface in which the make system interacts with the parsed source. The methods that must be implemented by the root class are,

```
virtual void enumerateMethodNames(AdtStringList& rMethodList) const = 0;
virtual bool hasClass(const char* pClassName,
                      string& rParentClassName) const = 0;
virtual bool flattenClass(const char* pClassName,
                          const AdtParserPtrList& rRootList,
                          string& rUsesList) = 0;
virtual bool optimise(const AdtStringList& rNewMethodList,
                      const AdtStringByStringMap& rNewMethodMap) = 0;
virtual bool expandMacros() = 0;
virtual bool extractClassConstructors(AdtStringList& rConstructorList,
                                      const char* pClassName,
                                      AdtSourceFileType nAsFileType) const = 0;
virtual AdtFile& writeClassMethodsAsFortran(AdtFile& rOutFile,
                          const char* pClassName,
                          const char* pLastClassName,
                          bool bExpandMacros) const = 0;
virtual AdtFile& writeNative(AdtFile& rOutFile) const = 0;
virtual AdtSourceFileType sourceFileType() const = 0;
```

The intentions of these methods can and should be determined through studying the existing code implementations. I will not go into any details here.

The *optimise()* method need not have a non-trivial implementation. This entry point is a place whereby loop array referencing optimisations can be implemented for Pascal. The implementation is currently a null implementation that does nothing, meaning ther are no Pascal loop optimisations. C++ does not require any such optimisations as the C++ compilers available generally already do this, so there is no inherent benefit to implementing it. For any new language support the remaining methods require proper implementation. This applies to the laguages already supported, except in the case of Fortran, which only gives non-trivial implementations to *writeNative()* and *sourceFileType()* (and the java support which is a skeleton implementation that isn't complete and probably wont be unless a java convert steps in and offers to push for it). This is the case because Fortran is simply used in managing the working files and is not used as a project source language.

4.3.2 Pascal Parser

The Pascal parser is implemented through the source files *delphi_l.l*, *dephi_y.y*, *delphi_inc.h*, *delphi_l_wrap.c*, *delphi_y_wrap.c*, *adtdelphi.hpp* and *adtdelphi.cpp*.

The lexer has two states, INITIAL and NOT_AD_CODE. The states exist to implement the *$ifdef $endif* conditional compilation of Pascal code. When the code is compiled into the parse tree it is in the INITIAL state and when it is being bypassed it is in the NOT_AD_CODE state. The conditional compilation handling is implemented using a fixed size stack with a maximum depth of 64. If there are more than this many levels of nesting of *$ifdef $endif* blocks then the compilation will fail.

Similarly, the lexer uses a fixed size stack to implement *$include* processing and has a maximum depth of 64. If more that 64 nested includes occur then compilation will fail.

The grammar has one additional non-terminal not part of the language specification, *Macro*, which is there to be able to compile the macro expansions that are defined in the *pascal_macros.txt* file. The rules in that non-terminal with trailing semicolons create a reduce-reduce conflict with the other parts of the grammar but this conflict can be safely ignored.

Macro expansion is handled in a rather complex manner by direct interpretation and manipulation of the parse tree. A simpler implementation could be constructed through the use of textual code construction parsed by the parser to produce replacement code fragments to replace the expression being expanded, as is done for the C++ case. This implementation is a legacy of my first attempt and works so will remain for the foreseable future. If problems arise with it then it should be re-implemented along the suggested line.

The class *AdtDelphi* implements the *AdtCompileBase* specialisation for the support of Pascal code and can be found in the *adtutils.hpp* and *adtutils.cpp* source files.

4.3.3 C++ Parser

The C++ parser is implemented through the source files *cpp_l.l*, *cpp_y.y*, *cpp_inc.h*, *cpp_l_wrap.c*, *cpp_y_wrap.c*, *adtcpp.hpp* and *adtcpp.cpp*.

The lexer has two states, INITIAL and NOT_AD_CODE. The states exist to implement the *#ifdef #endif* conditional compilation of C++ code. When the code is compiled into the parse tree it is in the INITIAL state and when it is being bypassed it is in the NOT_AD_CODE state. The conditional compilation handling is implemented using a fixed size stack with a maximum depth of 64. If there are more than this many levels of nesting of *#ifdef #endif* blocks then the compilation will fail.

Similarly, the lexer uses a fixed size stack to implement *#include* processing and has a maximum depth of 64. If more that 64 nested includes occur then compilation will fail. For C++ code *#include* macros are only processed if the include source file is enclosed in quotes. If enclosed in $< >$ the include operation is ignored. This is a deliberate design feature to allow ADT to ignore system includes which in all likelihood, will cause compilation to fail, as ADT only implements a restricted subset of the C++ grammar.

The grammar has one additional non-terminal not part of the language specification, *macro*, which is there to be able to compile the macro expansions that are defined in the *cpp_ macros.txt* file.

Macro expansion is handled through the use of textual code construction parsed by the parser to produce replacement code fragments to replace the expression being expanded.

The class *AdtCpp* implements the *AdtCompileBase* specialisation for the support of C++ code and can be found in the *adtutils.hpp* and *adtutils.cpp* source files.

C++ also requires complex handling of symbol tables and symbol scopes. This is implemented by the *AdtCppScopeManager* and *AdtCppScope* classes found in the *adtutils.hpp* and *adtutils.cpp* source files.

4.3.4 Fortran Parser

The Fortran parser is implemented through the source files *fortran_ l.l*, *fortran_ y.y*, *fortran_ inc.h*, *fortran_ l_ wrap.c*, *fortran_ y_ wrap.c*, *adtfortran.hpp* and *adtfortran.cpp*.

The grammar has one additional non-terminal not part of the language specification, *xCallExpand*, which was added to be able to compile the CALL expansions that were defined in the *fortran_ macros.txt* file, which has now been abandoned. Instead, the necessary conversions of PUSH and POP calls are now handled with a hard coded solution in the ADT source code.

The class *AdtFortran* implements the *AdtCompileBase* specialisation for the support of Fortran code and can be found in the *adtutils.hpp* and *adtutils.cpp* source files.

4.3.5 Automatic R Interface Code Infrastructure

The code responsible for the interpretation of automation commands and generation of R interface and module initialisation code is located in adtautomate.hpp and adtautomate.cpp. The code is structured around the concept of having to build interfaces for classes (*AdtAutoClass* class), each class having scalars (*AdtAutoScalar* class), arrays (*AdtAutoArray* class) and functions (*AdtAutoFunction* class).

As much as possible of the code generation code is shared within the *AdtAutoAttribute* and *AdtAutoHelper* classes.

The interpretation of array size expressions is handled by a special parser, the *AdtExpressionCompiler* class, which is implemented in the files *expression_ l.l*, *expression_ y.y*, *expression_ inc.h*, *expression_ l_ wrap.c*, *expression_ y_ wrap.c*, *adtexpression.hpp* and *adtexpression.cpp*. This allows us to evaluate and check the validity of array size comment expressions used in AD code.

Chapter 5

Using ADLib and ADLibPascal

The ADT package (automatic differentiation via TAPENADE) utility has a companion library to implement simplified and efficient array programming. The arrays it creates and manages can have up to 10 dimensions with user defined starting indices and sizes. The arrays are allocated in a single, flat and contiguous block of memory to aid in improved computational performance and easier interfacing with R arrays.

Arrays can be created independently or managed within an *AdtArrays* class. You will typically use the library by declaring your own class which publically inherits from *AdtArrays* and then implement the required numerical computations through class methods.

5.1 Array types

Multi-dimensional Arrays can be created for any of the (commonly used) intrinsic data types within the C/C++ language and most of the intrinsic data types in the Pascal language. To simplify coding and readability, *ADLib* and *ADLibPascal* define aliases for the different size and type arrays. The general format of the type name follows the pattern,

```
ARRAY_{number of dimensions}{type characters}
```

The number of dimensions can be between 1 and 10 inclusive and the type characters map to intrinsic types as in the following table.

C++ type	Pascal type	type characters
bool	*boolean*	B
longbool	*longbool*	LB
char	*shortint*	C
unsigned char	*byte*	UC
int	*integer*	I
unsigned int		UI
short	*smallint*	S
unsigned short	*word*	US
long	*longint*	L
unsigned long	*longword*	UL
float	*single*	F
double	*double*	D

For example, if we want a 3 dimensional array of doubles then the type would be *ARRAY_3D*. Note that for the Pascal language case unsigned integers are not supported because there is no unambiguous type for 32 and 64 bit platforms and therefore is not supported.

5.2 Creating Arrays

Arrays can be created in one of two ways, either with explicitely defined size/shape using an *AdtArrayPlan* instance (or the complimentary static *create()* functions), or implicitely by copying the shape of an existing array.

Explicitely specifying the array size and base indices is done through an *AdtArrayPlan* instance. The constructor is overloaded to cater for different dimension arrays and follows the generalised form,

```
AdtArrayPlan(int nBaseIndex1, int nSize1,
             int nBaseIndex2, int nSize2,
             int nBaseIndex3, int nSize3,
              .       .           .
              .       .           .
              .       .           . );
```

in the C++ case and,

```
AdtArrayPlan.create(int nBaseIndex1, int nSize1,
                    int nBaseIndex2, int nSize2,
                    int nBaseIndex3, int nSize3,
                     .       .           .
                     .       .           .
                     .       .           . );
```

in the Pascal case and supports up to a maximum of 10 dimensions. To create an array of that form with that size we call the *create()* method. As an example, lets say we want to create a series of 6 by 4 arrays whose indices are one based. We create an *AdtArrayPlan* instance that describes the array size with,

```
AdtArrayPlan SixByFour(1,6,1,4);
```

in the C++ case and,

```
Var
  SixByFour :AdtArrayPlan;
 .
 .
 .
SixByFour := AdtArrayPlan.create(1,6,1,4);
```

In the Pascal case. Then we call the *create* method to make the array with,

```
ARRAY_2D Array1 = 0;
ARRAY_2D Array2 = 0;
ARRAY_2I Array3 = 0;

SixByFour.create(MemAllocator, Array1);
SixByFour.create(MemAllocator, Array2);
SixByFour.create(MemAllocator, Array3);
```

In the C++ case and,

```
Var
  Array1, Array2, Array3 : ARRAY_2D;
 .
 .
 .
Array1 := nil;
Array2 := nil;
Array3 := nil;

SixByFour.create(MemAllocator, Array1);
SixByFour.create(MemAllocator, Array2);
SixByFour.create(MemAllocator, Array3);
```

In the Pascal case. The above code creates three arrays, six by four in size, whose base index is one. *Array1* and *Array2* are arrays of *double's* and *Array3* is an array of *int's*.

As an ease of use simplification the library contains overloaded static *create()* functions which internally create or access a cached array plan of the correct structure and use it to create the array (in the Pascal case we use global functions because static methods still require a this pointer to be called). In short it combines the above two steps into one but caches plans of a given structure to avoid creating multiple instances of plans with the same dimensions. Using this simplification the above arrays can also be created with,

```
ARRAY_2D Array1 = 0;
ARRAY_2D Array2 = 0;
ARRAY_2I Array3 = 0;

AdtArrayPlan::create(MemAllocator, Array1, 1, 6, 1, 4);
AdtArrayPlan::create(MemAllocator, Array2, 1, 6, 1, 4);
AdtArrayPlan::create(MemAllocator, Array3, 1, 6, 1, 4);
```

for the C++ case and,

```
Array1 := nil;
Array2 := nil;
Array3 := nil;

AdtArrayPlan_create(MemAllocator, Array1, 1, 6, 1, 4);
AdtArrayPlan_create(MemAllocator, Array2, 1, 6, 1, 4);
AdtArrayPlan_create(MemAllocator, Array3, 1, 6, 1, 4);
```

for the Pascal case. Note that we must supply an instance of the class *Adt-MemAllocator* to do the memory allocation for the arrays. It is structured this way to facilitate simpler management of arrays and their life cycle because if we fail to free the arrays explicitely they will be freed when *MemAllocator* is freed. Similarly, *MemAllocator* knows all about the arrays it creates so it can then easily create a same shaped array from any that it has previously created without the need of a plan.

If the allocation code above appears in a method of your class derived from *AdtArrays* then you will not need to create a *AdtMemAllocator* instance as the *AdtArrays* class has its own instance named *MemAllocator*. The class *AdtArrays* has simpler equivalent create methods so within that context the above example simplifies to,

```
ARRAY_2D Array1 = 0;
ARRAY_2D Array2 = 0;
ARRAY_2I Array3 = 0;

create(Array1, 1, 6, 1, 4);
create(Array2, 1, 6, 1, 4);
create(Array3, 1, 6, 1, 4);
```

in the C++ case and,

```
Array1 := nil;
Array2 := nil;
Array3 := nil;

create(Array1, 1, 6, 1, 4);
create(Array2, 1, 6, 1, 4);
create(Array3, 1, 6, 1, 4);
```

In the Pascal case. To create a same shaped array of an existing one we call the static create method of *AdtArrayPlan* with a parent array as the third argument in the call. Note that the parent array must be of the same type as the copy. For example,

```
ARRAY_2I    Array4 = 0;

AdtArrayPlan::create(MemAllocator, Array4, Array3);
```

in the C++ case will create *Array4* with the same plan as *Array3*. In the Pascal case the same would be achieved with,

```
Var
  Array4 : ARRAY_2I;
 .
 .
 .
Array4 := nil;

AdtArrayPlan_create(MemAllocator, Array4, Array3);
```

If you are writing code within a method of your derived *AdtArrays* class then we can use the simpler equivalent create method with the call,

```
create(Array4, Array5);
```

which applies equally to C++ and Pascal coding.

5.3 Mapping Memory as an Array

There is one other way we can create arrays and that is creating a map into a block of memory to be accessed as an array. The typical usage might be that you have a block of memory representing a 9 by 8 element matrix that has been passed to you from R and you want to index it in a simple fashion (R data structures are all simple blocks of memory irrespective of the dimension). To do so, we create a plan that fits the structure of the memory being passed to us and then pass an extra parameter in the create call, which is the pointer to the existing block of memory. For example,

```
AdtArrayPlan    NineByEigth(1,9,1,8);
ARRAY_2D        AliasArray = 0;

NineByEight.create(MemAllocator, AliasArray, (void*)pR_ArrayMemory);
```

shows the implementation in C++ whilst,

```
Var
  NineByEigth : AdtArrayPlan;
  AliasArray  : ARRAY_2D;
 .
 .
 .
AliasArray  := nil;
NineByEigth := AdtArrayPlan.create(1,9,1,8);

NineByEight.create(MemAllocator, AliasArray, pchar(pR_ArrayMemory));
```

show an equivalent Pascal implementation.

5.4 Ragged Arrays

In certain problems we may have multi-dimensional arrays where many elements are zero and need not be stored. In such circumstances being able to create ragged arrays may be helpful. ADLib allows the outer dimensions of an array to be ragged (vectors with different lengths) though creating a plan for such arrays is a little more involved.

As an example, lets say we wish to create a two dimensional array which comprises two vectors of different lengths, one 5 elements in length and the other 25. Furthermore we will use zero based indexing. The following C++ code fragment creates an array plan matching these requirements.

```
AdtArrayPlan    Ragged;
int             BaseIndices[2] = {0, 0};
int             Sizes[2] = {5, 25};

Ragged.beginRagged(0, 2);

for (int cn = 0 ; cn < 2 ; cn++)
{
  AdtArrayPlan  InnerArrayPlan(BaseIndices[cn], Sizes[cn]);

 Ragged.addArrayPlan(InnerArrayPlan, cn);
}

Ragged.endRagged();
```

To make a ragged column we need to call the *beginRagged()* method of the plan we are adding the column to, specifying the base index and the size of that column. This can be seen from the method prototype,

```
bool beginRagged(int nBaseIndex, int nSize);
```

in C++ and,

```
function beginRagged(nBaseIndex,nSize:longint):boolean;
```

in Pascal, which returns true if successful. After calling *beginRagged()* we need to loop through all the indices of the column and add array plans that specify the base index and size of that column entry using the *addArrayPlan()* method whose prototype is,

```
bool addArrayPlan(AdtArrayPlan& radtdiffArrayPlan, int nIndex);
```

in C++ and,

```
function addArrayPlan(rAdtdiffArrayPlan : AdtArrayPlan; nIndex : longint):boolean;
```

in Pascal. After specifying plans for all column entries we finalise the plan by calling the *endRagged()* method. It is essential that *beginRagged()* is always paired with a corresponding *endRagged()* call.

After *endRagged()* has been called arrays conforming to that plan can be created in the manner already outlined. These ragged arrays will be allocated as one contiguous block of memory irrespective of the array shape. Similarly, array mapping of existing blocks of memory works in the same manner as for conventional arrays.

5.5 Destroying Arrays

If you wish to discard an array that is no longer needed and reclaim the memory on the heap you can do so as follows. Either call,

```
AdtArrayPlan::destroy(MemAllocator, Array4);
```

in C++ and,

```
AdtArrayPlan_destroy(MemAllocator, Array4);
```

in Pascal, or if in the context of an *AdtArrays* derived class method then call,

```
destroy(Array4);
```

in both C++ and Pascal code.

5.6 Using the Arrays

Once created the arrays are simply used in the same manner as conventional n-dimensional C arrays or multi-dimensional arrays in Pascal. As an example, the C++ code fragment below calculates the average of all the elements in our 9 by 8 array passed from R above.

```
int    cn;
int    cm;
double dAverage = 0.0;

for (cn = 1 ; cn <= 9 ; cn++)
{
  for (cm = 1 ; cm <= 8 ; cm++)
  {
    dAverage += AliasArray[cn][cm];
  }
}

dAverage /= (9 * 8);
```

The equivalent in Pascal is,

```
Var
  cn, cm   : longint;
  dAverage : double;
 .
 .
 .
dAverage := 0.0;

for cn := 1 to 9 do
begin
  for cm := 1 to 8 do
  begin
    dAverage := dAverage + AliasArray[cn][cm];
  end;
end;

dAverage := dAverage / (9 * 8);
```

Note that in convetional C/C++ code, array bases are traditionally always zero, whereas the base index in our case is determined by the plan that created the array. Also note that the base indices you specify when creating the plan can be negative. This is made possible through pointer offseting and look up tables. As operator overloading is not employed there is no indexing overhead and no index bounds checking so it is imperative that care is taken not to use out of bounds indices.