Adding Fortran 90/95 Support and other Improvements to OpenFGG

Joakim Hjertstedt

June 17, 2006
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Robert Granat
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

The Fortran programming language is still an important programming language when it comes to numerical computing, in particular linear algebra. Although Fortran routines are fast, it can be cumbersome to test and develop large libraries of Fortran routines since the language lacks easy ways to create, modify and visualize the test data. This is where the OpenFGG (Open Fortran Gateway Generator) application comes in. It utilizes MATLAB:s MEX tool to create gateways between MATLAB:s environment and Fortran routines. This makes it possible to take advantage of MATLAB:s superior capabilities of modifying and visualizing the test data while retaining the speed of the Fortran routines without modifying them. Although OpenFGG has already proven to be a useful and valuable tool it lacks some functionality, most notably support for the newer Fortran 90/95 standard. This and other improvements to OpenFGG is the purpose of this Master's Thesis project.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Fortran is still the predominant programming language in numerical computing and seems to maintain its position also in the future. The foremost reason of this is the speed that can be gained from writing a program in Fortran but today another reason is the long tradition of developing library-standard numerical software in this language. There are, however, some disadvantages with developing routines in Fortran, most notably those for matrix computations. Construction of test examples requires the additional work of making test programs and using matrix generators. When switching from one test example to another, parts of the test program may have to be rewritten. It is also difficult to visualize the generated matrices with Fortran's text formatting capabilities.

By using MATLAB as a test platform one can solve these problems. In order to do so, one has to utilize the cross-language functionalities in MATLAB by using MEX-files. A MEX-file serves as a gateway between MATLAB and another programming language, which in this case is Fortran. Often, MEX-files are written by hand which can be quite time-consuming. This calls for the need of tools that can automatically generate them. OpenFGG (Open Fortran Gateway Generator) is such a tool that has been developed at the department of Computing Science at Umeå University. This application can parse Fortran source files for information about procedures' type and their input and output arguments. This information is then written to an XML-file and is also displayed in a graphical user interface. The user can, after the parsing process has finished, choose to change the input- and/or output-modes of the arguments before generating the gateway. Finally, the MEX-file and the Fortran source files are compiled into a library-file that MATLAB can use for calling Fortran routines.

Even though OpenFGG has proven to be a useful tool, it has had some drawbacks like the lack of support of the newer Fortran 90/95 standard. Although many library-routines still use the older Fortran 77 syntax, which is previously supported by OpenFGG, the newer standard is becoming more commonly used. Adding such support to OpenFGG would be greatly useful to the program's usability.

1.2 Outline

This report will have the following outline:

Chapter 2 describes the problem and defines the task of the project.

Chapter 3 gives a description of the Fortran language and its issues concerning this project.

Chapter 4 covers the tools and methods for accomplishing the project as well as some of OpenFGG's design.

Chapter 5 will describe testing and show a test example.

Chapter 6 summarizes the project and describes the restrictions of the system and suggests possible solutions.

Appendix A gives some notes regarding how to use the new version of OpenFGG.

Appendix B contains an updated document type definition (DTD) for OpenFGG.

Chapter 2

Problem Description

The purpose of this project is to develop a new improved release of the OpenFGG application. OpenFGG has previously been developed by Magnus Andersson [1] (core functionalities of the application like parsing and gateway generation) and Johan Sejdhage [2] (GUI). Both have developed the program as a Master's Thesis project at Umeå University.

Although OpenFGG has already proven to be a useful tool, its previous release had some restrictions. The probably most severe restriction was the lack of support of Fortran 90/95 syntax. The new standard of Fortran 90/95 also offers new functionalities that OpenFGG can support, like

- The INTENT statement which states a dummy argument's [1] input/output-mode.
- The OPTIONAL statement which states if a dummy argument can be left out when calling a procedure.
- Support of recursive procedures.
- A new less strict format for the source code called *free source form* as opposed to the older *fix source form*.
- New syntactic properties like declaring attributes for data types, support of more than one statement on the same line and longer identifier names with underscores, to name just a few.

Although Fortran 90/95 offers other new features, many have been deemed as unnecessary for the success of OpenFGG and will not be dealt with in this project. Besides the implementation of support of Fortran 90/95, this project will also deal with some other features that the previous release lacked like complete support for declaring arrays with ranges by the colon-notation (:), some bug fixes and some new GUI-related functionalities used for convenience.

[1] This is normally referenced as a formal argument in other programming languages.

Chapter 3

The Fortran Language and its Parsing Issues

Amongst programming languages Fortran really stands out as one of the earliest high level programming languages and certainly as the most persistent. Although other newer languages have taken ground from Fortran it is still a widely used language in certain areas like numerical computing. There are a lot of features and oddities of the Fortran language one has to consider when dealing with it, especially for parsing. This chapter will describe and discuss these issues and the most important of the new features in Fortran 90/95 as well. Since Fortran has such a historical legacy, it is best to start at the beginning...

3.1 The History of Fortran

Although Fortran is often credited as being the first compiled high-level language, there were some pioneering forerunners. These languages, however, never became widespread outside the place of their development like Fortran did.

Fortran was originally developed at IBM for their new IBM 704 system. This system which was introduced in 1954 offered the novelty of floating-point instructions in hardware. Before the introduction of this system, floating-point computations had been simulated in software which was very processor-consuming. Many of these old systems were also interpretive since this overhead was insignificant compared to the floating-point processing [3]. The introduction of IBM 704, however, prompted for the development of a new compiled high level language.

The development of the Fortran language was conducted by a team led by John Backus. The work began in 1954 and by the end of the year a report was published entitled "The IBM Mathematical FORmula TRANslating System: FORTRAN". It described the very first version of Fortran and it promised that it would achieve the same efficiency as hand-coded programs and also be as simple to use as interpretive systems. The claim of efficiency was first regarded with skepticism but when the first implementation of a compiler for the language was released in 1957 it proved that machine-code produced by the compiler was nearly as efficient as hand-written code [3].

Fortran soon became a very popular language due to its speed and ease of use. Another new advantage of the language was the ability to port programs between different

machines. This advantage resulted in Fortran being spread outside of IBM. Different implementations, however, often tended to use slightly different dialects which prompted for an effort of standardizing the language. The first standard document was released in 1966 and is known as Fortran 66 [4]. Later in 1978 a new standard was released called Fortran 77, a standard which is still commonly used. Work continued on the development of Fortran even further in the 80-s, work that some people feel was filled with political controversy that delayed the next release of the standard [5]. It was finally released in 1991 and is known as Fortran 90. The difference of this release compared to previous releases is that Fortran 90 is more a development of the language itself while others have been concerned with standardizing vendor-specific extensions [4]. The latest release of the language is called Fortran 95 and was released in 1997 and contains some minor improvements upon the Fortran 90 standard.

This project will be mostly concerned with the new features of Fortran 90/95 standard but also to some degree with the older Fortran 77 standard. Even though the Fortran language has evolved from its first release, many old constructs and restrictions, which were necessary due to the limited hardware capabilities of yesterday, are still supported by newer standards. This can complicate things a lot when parsing Fortran code. This will be covered in the following subsections.

3.2 General Parsing Issues with Fortran

There are several things that make Fortran special to other common high level languages in use today. These are very important issues to consider when parsing Fortran code.

The thing that is perhaps most striking is that whitespace is mostly not treated as significant. This can lead to very unreadable code if one is not careful. The consequence for the parsing process is that it requires the extra overhead of removing all whitespace when a Fortran source code-line is parsed for the first time. Also, there are no clear token boundaries when whitespace is insignificant and thus care must be taken when analyzing source code and separating between keywords and identifiers.

Another feature of the Fortran language that is of great concern when parsing Fortran code is that keywords are not reserved. The consequence for the user of the language is that one can use a keyword like `IF` as an identifier. This freedom comes however at a price. First, it makes it possible to write less readable code and second, it makes the parsing process more difficult since keywords are only significant in certain contexts. To solve this the parser must switch between different states when parsing different parts of a source code-line since a keyword may be applicable at one point but not at another.

A commonly used example of Fortran text books to demonstrate these properties are the following two lines of Fortran code:

```
DO 10 I = 1.5
DO 10 I = 1,5
```

The first line assigns the value `1.5` to a variable named `DO10I`, the second line is a do-statement. Note that it is only the comma and the punctionation mark between `1` and `5` that distinguishes the two statements from each other.

One thing which is also troublesome when parsing Fortran code is the support of implicit typing rules. In the first version of Fortran, all types were implicit. This meant that variables beginning with the letters `I`, `J`, `K`, `L`, `M` or `N` were considered as integers, while other variables were considered as floating point numbers [3]. For some reason

this feature has never been removed from Fortran. This means that compilers will not generate an error if a type declaration is left out, but will instead enforce the implicit typing rules. In fact, later Fortran standards have included the ability for the user to create their own rules for implicit typing using the `IMPLICIT` statement. Fortunately, for readability's sake, implicit typing rules can also be turned off, using the `IMPLICIT NONE` statement at the beginning of a code block. When this statement is used the compiler will generate an error if a type declaration is left out.

Another important feature of the Fortran syntax is that the source code is divided into lines. A line can be a comment line, an initial line that contains the beginning of a statement or a continuation line where a statement continues from a previous line. Determining which type a line belongs to is usually the first step when parsing Fortran code.

There also exists some other, minor, issues with the language like upper- and lowercase letters not being significant in the code, except for string literals.

3.3 New Features of Fortran 90/95

As stated previously in this report, Fortran 90 was a huge development of the Fortran language. It contains lots of new features as well as newer syntax forms. One of the main reasons for developing Fortran 90 was to add features that other newer high level languages had in order to compete with them. Although much was added, nothing was removed from the previous Fortran 77 standard, but some features were labeled as *deprecated*. As a result of this, the Fortran 90 language is huge and many of the things one can do in the language can be done in at least two ways. In the Fortran 95 standard some of the features that were deprecated in Fortran 90 have been removed. Most of the newer Fortran compilers, however, still support the old features. Thus, in order for a program dealing with Fortran 90/95 to be successful, it ought to be able to cope with both old ways and new ways of writing code.

In the following subsections, some of the most important of the new features will be described. Facts and examples have been taken from [4], [6] and [7].

3.3.1 Comments

A new feature in Fortran 90/95 is that one can denote a comment by using an exclamation mark '`!`'. Everything to the right of the exclamation mark is regarded as a comment. That makes it possible to have some statement at the beginning of a line and a comment to the right of it.

In contrast, the old Fortran syntax uses the characters '`C`' or '`*`' to denote a comment. The character must be placed in the first column of the line and the entire line will be treated as a comment.

3.3.2 Continuation Lines

In older Fortran syntax, a continuation line is denoted with an arbitrary character, with the exception of a zero or blank space, which is placed in column 6. This will look something like this:

```
 X = 5 + 3
$ - 1
```

where `$` is the arbitrary character.

Fortran 90 introduces a new way of denoting a continuation line by using the ampersand character '`&`'. This character is, however, placed at the end of the line which will be continued on the next line. Optionally, an ampersand can also be placed as the first character at the continuation line as well. As opposed to the old way of coding the ampersand can be placed in any column. The same statement as above can, by using ampersand, be written like:

```
X = 5 + 3   &
      &  - 1
```

3.3.3 Several Statements on the Same Line

In Fortran 90, not only can one continue a statement over several lines but one can also write several statements on the same line. The semicolon character, '`;`', is used as the delimiter.

3.3.4 Free Source Form

Code written in previous standards of Fortran has been restricted to a specific format. This format has its roots from the days when computer programs were printed onto hard-copy cards, where certain code should be placed in certain columns.

In classical Fortran, columns 1-5 are reserved for line-labels, that is a number from 1-99999. Column 6 is reserved for an eventual continuation line-character and the columns to the right of it are used for ordinary program code. This form of source code, which is called *fix source form*, is still widely used today.

With Fortran 90, a new source form has been introduced, called *free source form*. As the name implies, it removes the restrictions of the fix source form. In free source form, source code can appear most anywhere on a line. Line-labels must of course come before other code segments but are not restricted to columns 1-5. A special character, indicating continuation lines has also been introduced which is not restricted to column 6 (see above). Another new feature of the free source form is that whitespace is treated as being significant.

There are, however, some problems with introducing a new source code format. Since all of the previous Fortran code has been written in fix source form, the user community of Fortran has not been eager to start using the free source form immediately. When coding, it is also important to take care and not mix code which is compatible with one source form but not the other. Fortran compilers tend to, by default, parse source code files with the file extensions `.f90` and `.f95` in free source form and other files in fix source form. There exists, however, a method for writing code that can be used in both source forms (see page 79 in [6] or page 2-16 in [7]).

3.3.5 Named Blocks

Another new feature of Fortran 90/95 is the ability to name blocks like if-blocks. One can also use the name at the end-statement of a block.

3.3.6 Type Attributes

In Fortran 90 the programmer has the option of stating attributes connected to variables in a type declaration statement, instead of having to write several separate statements. Consider the following statements that are valid in Fortran 90/95 as well as previous Fortran releases:

```
INTEGER X, Y
DIMENSION(2,2) X, Y
```

In Fortran 90/95 the same can be written as:

```
INTEGER, DIMENSION(2,2) :: X, Y
```

Note, that when using attributes in a type declaration, the double-colon operator, '::', must be used. This operator can also, most of the time, be optionally used in declarative statements other than type declarations.

3.3.7 Recursion

One of the biggest news in Fortran 90 is the ability to use recursion in procedures. In previous Fortran releases, all of the memory was statically allocated at the start of a program. This results in very fast program execution but prevents recursion, amongst other things.

A recursive procedure must be declared with the `RECURSIVE` keyword. It must also be declared with a result-variable which is enclosed between paranteses and preceded by the keyword `RESULT`. As an example, a function that computes the faculty of an integer by using recursion is shown below:

```
RECURSIVE INTEGER FUNCTION FAC(N) RESULT(RES)
    IMPLICIT NONE
    INTEGER N
    IF(N<=1) THEN
        RES = 1
    ELSE
        RES = N*FAC(N-1)
    END IF
END FUNCTION FAC
```

3.3.8 The INTENT Attribute

The `INTENT` attribute is used to tell the compiler if a dummy argument is used as input, output or both. This can also be very useful for a program like OpenFGG which must have this information in order to produce a correct gateway.

3.3.9 The OPTIONAL Attribute

By declaring a dummy argument with the `OPTIONAL` attribute, the programmer tells the compiler that the argument does not have to be provided when a call to the procedure is made. In order for this to work, each such argument must be tested inside the procedure with the function `PRESENT` to determine if the argument has been provided and then take the appropriate action.

3.3.10 Allocatable Arrays

Arrays that can be allocated and deallocated is yet another new feature in the Fortran 90 language. To create an allocatable array one must declare it with the `ALLOCATABLE` keyword. To allocate the array the programmer uses the `ALLOCATE` statement and to deallocate the array the `DEALLOCATE` statement is used. One important thing to note is that dummy arguments cannot be declared as allocatable arrays although one may pass an allocatable array as an actual argument to a procedure.

3.3.11 Pointers

Another of the big news in Fortran 90 is the support of pointers. There are, however, some differences between the use of pointers in Fortran and in other languages like C. A pointer in Fortran is not its own data type but rather an attribute to a variable using the keyword `POINTER`. The pointer can point to any variable of the same type which has been declared with the `TARGET` attribute. A pointer thus acts as an alias of the target variable. Dummy arguments can be declared as pointers.

3.3.12 Derived Data Types

Fortran 90 also introduces derived data types. A derived data type is similar to a C struct, containing a set of variables that are of intrinsic or derived data types. A derived data type can have its variables initialized by enclosing the constants within paranteses which are preceded by the type name. Below is an example of how to use a derived data type in Fortran. The type `POINT` represents a two-dimensional point:

```
TYPE POINT
    REAL :: X, Y
END TYPE POINT

TYPE(POINT) :: P1
TYPE(POINT) :: P2

INTEGER :: DIFFX, DIFFY

P1 = TYPE(23.5, 5.6)
P2 = TYPE(42.1, -3.9)

DIFFX = P2%X - P1%X
DIFFY = P2%Y - P1%Y
```

3.3.13 Interfaces

Interfaces in Fortran are used to tell the compiler how the head of a subprocedure looks like, that is its type, arguments and the arguments' types. Sometimes an interface must be provided to use the new features of Fortran 90/95 but they can always be used optionally. An interface can also be used to overload some operators when declared inside a module block (see below).

3.3.14 Modules and Internal Procedures

A module block is a new feature in Fortran that enables encapsulation of data, interfaces and procedures. A module is somewhat similar to a class in an object oriented language. Its data can also be declared as private, if the data only should be accessible within the own module or public if the data should be accessible outside the module.

Internal procedures is something that can be declared in both module blocks and procedure blocks. These procedures have access to all the variables that have been declared in the block which contains the internal procedures. To declare internal procedures, the `CONTAINS` statement must first be declared inside the block.

3.3.15 Other Minor Additions

Fortran 90/95 also introduces some minor additions and improvements. The maximum length of identifiers has been increased to 31 characters. Also, identifiers now may contain underscores. The maximum length of a line has also increased from 72 to 132 characters.

Chapter 4

Tools, Design and Implementation

The task of adding new features to a full-scale program like OpenFGG, when one has not been part of the initial development, offers some great challenges to face and overcome. Most notably, one has to obtain a thorough understanding of the system and the problems it is designed to solve. The design decisions one has to take are sometimes very difficult. Something that at first glance seems to be a good solution to a problem, may later show up to be a really bad decision, as one learns more about the system.

In this chapter, the tools, design of the relevant parts of OpenFGG as well as the most important aspects of the implementation will be discussed and presented.

4.1 Previous Development of OpenFGG

As mentioned before, the OpenFGG application has been previously developed by Magnus Andersson and Johan Sejdhage as two separate Master's Thesis projects. Magnus Andersson has contributed to the core functionalities of the program, developing the ability to parse files with Fortran 77 source code and generate gateway-files that can be used by MATLAB. Johan Sejdhage's contribution to the project is the GUI features of the program and some other improvements like project handling and splitting complex-type parameters.

The application is a rather big system, consisting of about 15 000 lines of code in over 70 classes. It was quite clear from the beginning of the project that getting an understanding of this complex system would take some time. Also, the fact that the code is only sparsely commented and that there does not exist a complete class diagram of the system makes the task even more difficult. Inserting code in the right place, without messing up the program, can be somewhat compared to a surgical incision. There have, however, been much help received from the previous developers of the program, as well as from the project's supervisor.

For more details on Magnus Andersson's [1] and Johan Sejdhage's [2] work, see their respective Master's Thesis report.

4.2 Tools used in this Project

OpenFGG is written in Java. This makes it easy to port the program between different platforms. OpenFGG has been successfully tested both on Unix/Linux, which is the target platform, and Windows. It is also very easy to make a functional GUI in Java. Another advantage with using Java, which is a very safe language, is that it makes it possible to manage a system as large as this without the use of special development-environments and debuggers. The language has also lots of support libraries for parsing and XML processing which OpenFGG can make use of. The Java versions which support OpenFGG are 1.5 and later.

For the detailed parsing, the Java Compiler Compiler, JavaCC[1], library is being used. The programmer provides a specification file, containing a language grammar, to JavaCC which then generates a parser. This is a more flexible approach than to write a parser by hand since the programmer can concentrate on the language grammar while the implementation is left to JavaCC. It is also easy to build tree structures automatically when using JavaCC for parsing. This is especially useful when parsing expressions. The version of JavaCC used in OpenFGG is 3.2.

For XML processing, the JDOM[2] library is used which makes it easy to read from and write to XML-files. Another library being used is the Commons Math[3] library, which adds support for handling certain data types which Java does not support natively.

To build the OpenFGG system, a tool called Ant[4] has been used. This program makes it easy to setup rules when compiling, similar to the make utility found in UNIX/Linux environments.

Amongst the literature that have been used, the *Compaq Fortran Language Reference Manual* [7] has shown to be an invaluable asset, but other Fortran literature like [4] and [6] have also proven to be very useful.

4.3 Description of the Parts of the OpenFGG System Relevant to the Project

As stated before, the OpenFGG system is quite large and as such it is difficult to visualize it in a clever, compact way. The classes can, however, be divided into certain categories based on their functionalities in OpenFGG. In this section, the parts of the OpenFGG system most relevant to this project will be presented, even though some changes have been made to other parts as well.

4.3.1 GUI and General OpenFGG Functionality

There are three core classes in OpenFGG where every flow in the program will go through. The first of them, simply called `OpenFGG`, is the main class which is foremost used as a container class for the other two, `OpenFGG_GUI` and `FGG`. The `OpenFGG` class is also concerned with redirecting program control between `OpenFGG_GUI` and `FGG` through various class methods.

The `OpenFGG_GUI` class is, as the name suggests, where the GUI functionality of OpenFGG resides. There are several *Swing* components that have been used to build

[1] https://javacc.dev.java.net/
[2] http://www.jdom.org/
[3] http://jakarta.apache.org/commons/math/
[4] http://ant.apache.org/

the GUI. The class also contains the *listeners* which are used to respond to user input, often resulting in calls being made to methods in the instance of the `OpenFGG` class. `OpenFGG_GUI` is, however, a class of gigantic proportions containing well over 2000 lines of code. This can make it quite tideous and time-consuming to find the right code segment when adding new features to the system. Besides GUI components this class also contains objects of classes used for project handling.

The last of the core classes, `FGG`, is concerned with the core functionalities of OpenFGG, containing class objects representing the gateway and error handling. It is in the methods of this class where calls to more specific parts of OpenFGG are made.

4.3.2 Parsing

The classes in OpenFGG which are used in the parsing process are the ones that this project has been most concerned with. It is best to describe the parsing process from start to finish, giving more detailed descriptions of each class as they are used in the process.

The parsing process starts in, the previously described class, `FGG` by making a call to the method called `parse`. This method takes as input a string with the name of the file to be parsed. The result of the parsing process is a `Source` object which contains all information of each procedure found in the file. This object is then placed in the `Gateway` object of the `FGG` class instance, which stores all necessary information used for creating the gateway-file of the project.

The parse method in the `FGG` class calls, in its turn, a method called `parse` in the class `FortranParser` (known in previous versions of OpenFGG as `F77Parser`). This class contains static methods and data used for parsing each statement in the Fortran source code-file, separating the information contained within each procedure into `Procedure` objects that are added to a `Source` object.

Most of the parsing process takes place in the method called `processFile`. Here, each line in the source code-file is being read in a loop. The first task is to determine which type the line belongs to. Is it an initial line, a continuation line or a comment? This is done in the method `getLineType`. After this is done any eventual continuation line is appended to the code from the previous lines.

When a complete statement has been read, the statement is prepared and its type classified. This is done in the `StatementScanner` class. This is a very important class in the parsing process. Because whitespace and upper-/lower-case letters are not significant in classical Fortran, this class removes all of the whitespaces and also makes all source code-letters in the statement upper-case, except for string literals. It also handles some types of constants and line labels for further use. When these modifications have been made to the statement, the type of the statement is determined, most often by looking for a keyword at the beginning of the statement. The modified statement-text and other information associated with the statement is returned in a `Statement` object.

After the `Statement` object has been returned from the `StatementScanner` class, it is sent to the `ProcessStatement` method in `FortranParser`. In this method an appropriate action will be taken depending on the statement's type. If the statement needs to be parsed more thoroughly, it makes a call to a corresponding method in the `StatementParser` class which has been generated by JavaCC. Information like a variable's name, type and attributes are stored in the parser's `SymbolTable` object. Each identifier which is stored in the symbol table is represented by a `Symbol` object, an internal class of `SymbolTable`.

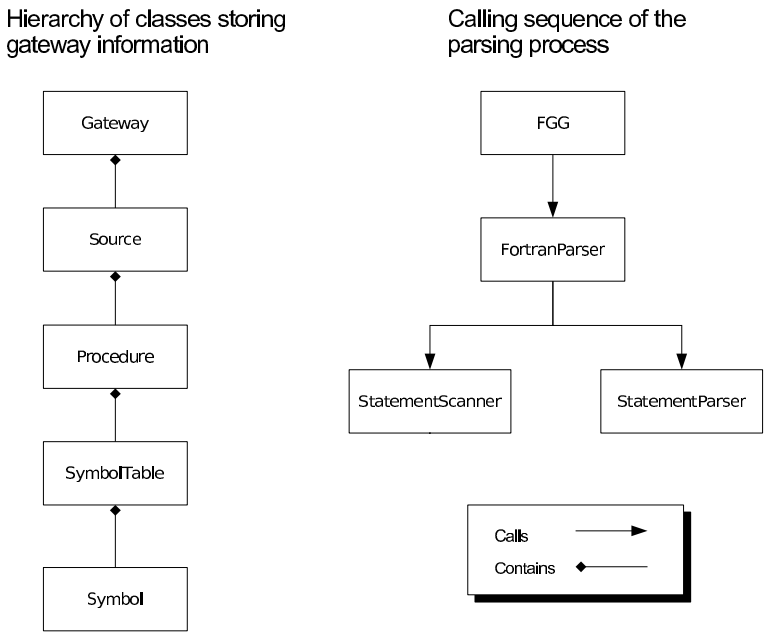

Figure 4.1: The parsing process and its classes in OpenFGG.

When an end-statement of a procedure is found, the `SymbolTable` and some other information are put in a `Procedure` object, which in its turn is put in the `Source` object of the source code-file. This concludes the parsing process.

Figure 4.1 shows an illustration of the parsing process and its classes.

4.3.3 XML-File Processing and Gateway Validation

When adding new features to OpenFGG, it is also often necessary to make some changes in the classes which handles the XML-file processing and the gateway validation. Validation of the correctness of the gateway is performed every time before the gateway is either exported or imported to the project's XML-file. The class which takes care of the validation is `GatewayValidator`.

The classes for importing and exporting the project's `Gateway` object from/to a file are, naturally, called `XMLImporter` and `XMLExporter` respectively. These classes utilize the JDOM support library for easy handling of XML input and output.

4.4 Implementation of the new Features

Most of the work has been spent on solving the parsing issues when adding Fortran 90/95 support. As Fortran 90 is a superset of Fortran 77 it is important to make the old syntax work together with the new additions in the newer syntax. In this section, the most important parts of the implementation will be described.

4.4.1 Statement Classification

Because the newer syntax is more complicated than the old Fortran 77 style, it soon became clear that the old techniques used by OpenFGG to identify the type of a statement were inadequate. Previously, an algorithm described in [1] had been used for statement classification. One of the most important aspects of this algorithm is the classification based on finding a keyword at the beginning of the statement. It is not, however, always as simple as to look at the beginning of a statement for a keyword to classify a statement in the Fortran 90/95 syntax since keywords can come in different orders for some statements. As an example, consider the syntax of a function declaration in Fortran 90/95, taken from the Compaq Fortran Language Reference Manual [7], where code enclosed by brackets is optional:

```
[prefix] FUNCTION name ([dummy-arg-list]) [RESULT (result-name)]

where prefix is one of the following:

type [keyword]

or

keyword [type]

where keyword is one of the following:

RECURSIVE or PURE or ELEMENTAL
```

As this example shows it is quite difficult to determine that a statement is a function just by looking for a keyword at the beginning of the statement. As a further note, a subroutine declaration in Fortran may also have the keywords `RECURSIVE`, `PURE` or `ELEMENTAL` at the beginning of the statement.

Another thing that complicates things is that blocks in Fortran 90/95 can have end-statements containing the type of the block, as well as the name of it, for example `END FUNCTION FUNC_NAME`, where `FUNCTION` is the type and `FUNC_NAME` the name of the block. The name of a block is always optional to provide in an end-statement but the type is sometimes required. Older Fortran standards only had the `END` keyword at the end of a procedure or program block which simplified the end-statement classification in OpenFGG, but in order to support Fortran 90/95 these things must also be considered.

To solve these problems the *pattern recognition* capabilities found in newer Java-versions have been used for statement classification and end-statement detection in the `StatementScanner` class. The pattern recognition is performed by the use of *regular expressions*, a great method for matching simpler patterns of text. A regular expression is compiled by calling the static method `compile` which is found in the Java-class `Pattern`. The method returns a `Pattern` object which then can be used for matching text with the pattern. If a match is found, a `Matcher` object, which is returned by the `Pattern` class's `matcher` method, can be used to extract certain parts of the pattern. This can be very useful, for example when extracting the name of a procedure block. Another good thing is that the string with the regular expression can be built by strings consisting of other regular expressions, making it easy to control the amount of detail

the expression should have. Below is an example, showing the regular expression of a function pattern:

```
"(.*)(FUNCTION(" + identifier + ")\\((" + argsList + ")?\\).*)"
```

It is somewhat simplified compared to the definition shown above, but demonstrates how regular expressions are used. The paranteses are representing a group in the regular expression which can be extracted by a `Matcher` object. For simplicity, the prefix, where '`.*`' stands for zero or any number of arbitrary letters, is not pattern-matched at this stage but is first extracted and then tested. The regular expression is concatenated with two other regular expressions, found in the strings `identifier` and `argsList`. From this pattern it is easy to extract the name of the function and create a new pattern which tests for the end-statement of the function block.

Another advantage with using `Pattern` and `Matcher` objects for statement classification is that the removal of unnecessary keywords from the statement after classification, can be greatly simplified. Before, the length of the keyword had to be hard-coded in order to remove it, but with the use of regular expressions one can simply specify the groups which should be kept. If the group number is the same over several patterns, this step can be generalized for those patterns.

For more information on how to use the `Pattern` and `Matcher` classes and regular expressions in Java, see the API specification [8].

4.4.2 Detailed Parsing

At one point in the development process, it was felt that perhaps the use of regular expressions could also have been used for most of the detailed parsing in OpenFGG as well. However, regular expressions do have their limitations, especially when it comes to parsing more complicated syntax like expressions. For detailed parsing, a parser-generator like JavaCC is a much better choice.

The use of JavaCC for parsing Fortran code is, however, far from problem free. The fact that Fortran does not reserve its keywords means that the parser must switch between states. In the default state, identifiers of every name, including keywords, may be parsed. In order to parse keywords that have been left from the statement classification process the parser must change to a special state where the keyword is treated as a token. Below is an example, taken from the JavaCC specification-file `StatementParser.jjt`, of a declaration for three types of states where the keyword `KIND` is treated as a token.

```
<KIND_STATE,KIND_OR_LEN_STATE,TYPE_STATE> TOKEN:
{
    <KIND: "KIND">:INT_STATE
}
```

The colon denotes a switch to another state, `INT_STATE`, which happens after the `KIND` keyword has been parsed. The constant switching of states impacts on the specification-file's readability when ordinary Java source code (a call is made to an object's method when switching between states) is mixed with grammar specifications. It is also important to write the code carefully in order to avoid unexpected parsing behaviour when switching from one state to another.

Another problem when parsing Fortran is that some things can be done in several different ways. This can make the parser unable to decide which branch of the grammar to take just by looking at the next token. In order to solve such problems it is often necessary to use the `LOOKAHEAD` operator in JavaCC. By using `LOOKAHEAD` it is possible to peek at tokens that comes after the current token, and by looking at them the parser can decide which branch to take.

Even though state switching and the need to use `LOOKAHEAD` seemed a little discouraging at first, one quickly gets the hang of it. Lots of well-needed help has also been provided by Magnus Andersson, the original programmer of the JavaCC specification-file.

A new class, called `Attributes`, has also been created to hold attributes when a type declaration is being parsed in `StatementParser`. Further, the internal class `Symbol` had to be modified in order to store new types of information, like the intent of a dummy argument or if an identifier is used as a result-variable.

4.4.3 Parsing of Non-Declarative Statements

In the previous version of OpenFGG, all of the statements in a Fortran source code-file were more or less parsed in detail, even non-declarative statements like executable statements. This meant that the source code file would be completely parsed two times whenever a change was made to the source code. Once by OpenFGG and once by the Fortran compiler used by MEX.

The foremost reason for parsing every type of statement was to check if a dummy argument was used as a dummy procedure, something that is not allowed in OpenFGG. It is, however, possible to declare that an argument is a dummy procedure by using the `EXTERNAL` statement or, in Fortran 90/95, declaring an interface.

Due to the fact that Fortran 90 introduces a lot of new program constructs, some which are somewhat complex to handle when parsing, it was decided by both the authour and the supervisor of this project that it would be unnecessary to parse every statement in detail. Therefore, the new OpenFGG version will only perform detailed parsing on declarative statements where information needed to build the gateway can be found.

The advantage is, besides the efficiency increase when parsing, that this will make it easier to add support of newer standards of the syntax, since not all of the syntax needs to be understood by the parser. A statement that cannot be classified is labeled as unknown and is ignored for the rest of the parsing process.

A disadvantage of this approach is less exact error reporting, since a faulty statement may go undetected during parsing. This is not, however, such a big problem since OpenFGG in this case will report that the gateway is faulty, or report an error which can be deduced from the faulty statement. Syntax errors will also be reported by the Fortran compiler used by MEX. Another disadvantage is that dummy procedures will only be detected by using the `EXTERNAL` statement or interfaces. It is, however, good programming practice when coding Fortran to always use these methods when declaring external procedures that are being used in a program, so it will probably not cause so much inconvenience.

4.4.4 Free Source Form

In order to support the new free source form, changes have been made to several classes. Most notably, the first parsing steps like determining the type of a line (initial, con-

tinuation or comment) and reading line labels work differently compared to the old Fortran style. By adding necessary code in the method `getLineType`, found in the class `FortranParser`, and adding new methods for detecting the ampersand continuation character and removing it, the line type-classification support was fixed. To read line labels which are not restricted to columns 1-5, code was added to the `readLabel` method of the `StatementScanner` class.

Another implemented addition is the ability for the user to change which source form OpenFGG should use when parsing a file. This has been achieved by implementing two radio buttons in the *Project Setup* window, which is shown in the GUI either when selecting `Setup` from the project menu or when creating a new OpenFGG project. By default, OpenFGG chooses the source form to be used by looking at the file's extension. When a file-name has the `.f95` or `.f90` extension, free source form is used, otherwise fix source form is used. By selecting a source file in the *Source files to parse* list, the user can manually choose which source form to use by pressing the respective radio button.

Another restriction that existed in the previous version of OpenFGG was that a line of source code was restricted to 72 columns. This was to conform to the standard of fix form. Since free source form supports 132 columns of source code, and many compilers support even more, this restriction has been removed completely from OpenFGG.

Chapter 5

Testing

The testing process has proceeded throughout the whole project. Making sure that a new feature works correctly, while older functionalities remain intact has been a vital part of the development of OpenFGG. Most of the tests have been conducted with smaller test programs, that explicitly tests a certain aspect of the program.

As a larger test example, a numerical software library called *RECSY*[1] has been used. RECSY has been developed at Umeå University and is used for solving triangular Sylvester-type matrix equations. It is also a library that has been written with Fortran 90 code, and uses features as INTENT, allocatable arrays and recursion.

The test platforms used for testing RECSY have had the following properties:

- SunOS 5.9
- f90 Fortran compiler
- MATLAB version 7.0 (R14)

and

- Windows XP Professional Edition
- Compaq Visual Fortran version 6.5
- MATLAB version 6.5 (R13)
- Precompiled, Visual Fortran version of *BLAS*[2] and *LAPACK*[3]

The testing has been successful and OpenFGG has been able to create a library file that can be used by MATLAB for calling routines found in RECSY. A new Fortran 90 feature that is very handy when compiling a software library as large as RECSY is the INTENT attribute. By parsing the INTENT attribute, OpenFGG can fill in the correct mode of an argument(input, output or both), something which the user has to provide in the GUI otherwise.

Figure 5.1 shows a screenshot of OpenFGG where the RECSY routines are being tested.

[1] http://www.cs.umu.se/~isak/recsy/
[2] http://www.netlib.org/blas/
[3] http://www.netlib.org/lapack/

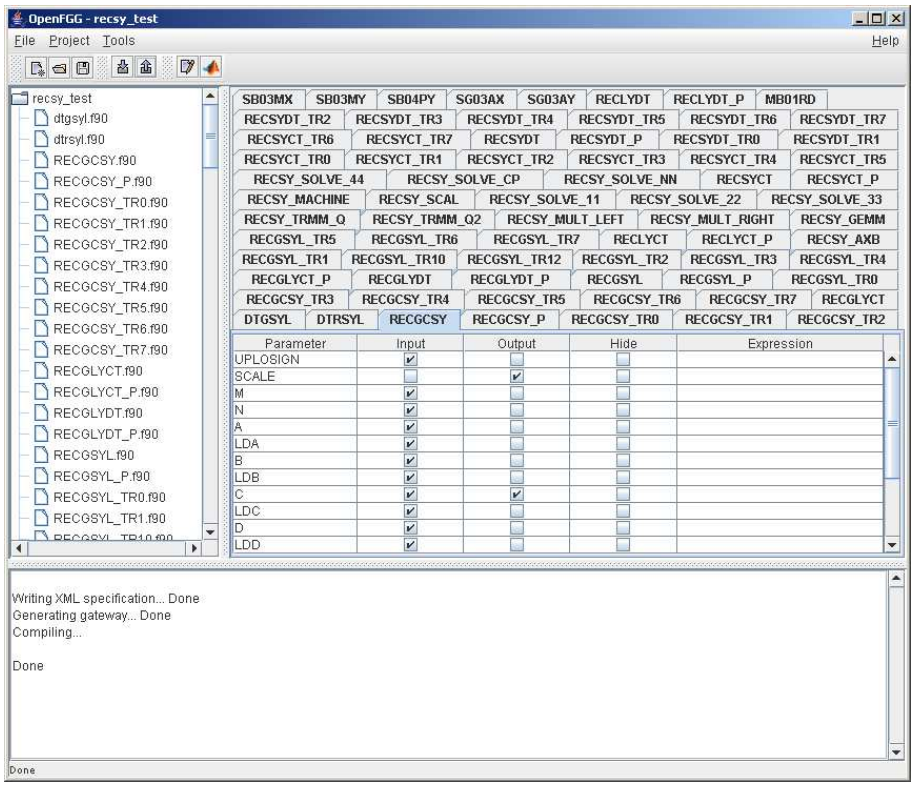

Figure 5.1: Screenshot from the testing of RECSY with OpenFGG.

Chapter 6

Conclusions

This project has been very interesting to undertake. To gather knowledge and understanding of a large system like OpenFGG is quite challenging and time-consuming, especially when the original developers are not available at your own working location. Something that, in the end, proves to be very easy to implement, may take several days or weeks to think out and design.

Most of the time of this project has been spent on information gathering and design issues. Another time-consuming aspect of the work has been testing, conducted in order to validate the correctness of the additions implemented in the program. The time spent on coding, however, has not been so great. It is the road to get there that takes time.

All in all, this project has been a very useful experience, where one has developed a program which one has not had any previous knowledge of. This is a task that is very common when it comes to software development. The experience of getting a deep understanding of a high-level language like Fortran has also been very good, despite the fact that Fortran is quite a quirky language.

6.1 Limitations and Future Work

Even though OpenFGG now is a very complete program, there still exists limitations, some which may be fixed in the future. In this section, the most important limitations of the program and what can be done for them will be discussed.

6.1.1 Modules and Internal Procedures

OpenFGG does not generate an error when encountering module blocks or internal procedures but simply skips them.

In order to parse internal procedures and add them to the gateway, the program must copy the symbols found in the symbol table, that have been parsed in the containing block, into a local symbol table. This is because of the fact that variables declared in the containing block are also visible in the internal procedures.

Something that must be added to OpenFGG in order for module support, is the ability to handle aliases for symbols. To use a module one declares a `USE` statement, where one can also specify a list of aliases for the symbols declared in the module to be used. Some sort of support for distinguishing between public and private variables is also probably required.

Support of internal procedures has not been deemed as a necessary thing to support at this time, but increased use of modules in software libraries may prompt for added support of them and their internal procedures in the future.

6.1.2 Assumed-Shape Arrays

In Fortran 90/95, the programmer can declare assumed-shape arrays. This means that the size of one or more of the dimensions used in the array are not specified. This is done with the colon-notation, for example:

```
INTEGER :: A(2, :, :)
```

This is a three-dimensional array where the first dimension is fixed, but the two others could be of variable size. A dummy argument can also be declared as an assumed-shape array, but the calling procedure is then required to provide an interface of the procedure to call.

OpenFGG does not currently support assumed-shape arrays as dummy arguments. In order for this to work, the `GatewayGenerator` must be changed so that it can write a necessary interface and use the colon-notation in the gateway-file. Some changes must also be made to the XML format and the XML import/export functionalities.

It was the authour's intention to add support for assumed-shape arrays, but due to time constraints this addition is left as future work.

6.1.3 Pointers

Pointers can be declared as dummy arguments in Fortran 90/95. It is, however, not clear how to map this type between Fortran and MATLAB. In MATLAB, the notion of pointers does not even exist.

Because of this reason, OpenFGG does not support pointers as dummy arguments, and they will most likely not be supported in the future.

6.1.4 Derived Types

It would probably be possible to add support for derived types in OpenFGG. A suitable data type found in MATLAB, which it could be mapped to, is `struct`. Support for derived types would, however, require a big effort of extending the processes of parsing and checking the data being copied between MATLAB and Fortran. This since a derived type may be nested by many other derived data types, including itself.

Derived types are, however, not so common when it comes to software libraries for numerical computing, so the support of this feature is probably not very necessary.

6.1.5 Dummy Procedures

Dummy procedures[1] are still not supported by OpenFGG. It is probably not possible to map a MATLAB function to a Fortran argument using MEX, and even if it did, it would be a considerable complicated process involving the creation of a reverse gateway, which would call the MATLAB function from inside the Fortran procedure.

[1] a procedure that is provided as an argument

Fortunately, dummy procedures are not so common in library routines for numerical computing and it has never been the purpose of OpenFGG to call MATLAB functions from Fortran procedures.

6.1.6 The OPTIONAL Attribute

OpenFGG can now parse the `OPTIONAL` attribute and write this information to the XML-file of the project. There are, however, some things left to consider of how this feature should be supported in OpenFGG. The ideal thing would be to make OpenFGG to have the ability to create gateways which can be called with, or without the optional arguments when called from MATLAB. The problem with this approach is how to make the gateway know which argument has been left out since MEX only can get information about the number of arguments and their data types, not the name of the Fortran dummy argument which an actual argument maps to.

Another approach would be to provide the ability for the user to choose, in the GUI of OpenFGG, if the gateway should be generated with the optional argument or not. It has, however, been shown to be troublesome to distinguish between optional and non-optional arguments in the GUI, in a clever way, without having to create a custom rendering class for this purpose.

Due to this and time constraints, the implementation of this feature has been left as future work.

6.1.7 Bugs

There are still some bugs left in the program that should be fixed in the future. As an example, sometimes when changes are made to a project and the user tries to exit the application, the program will not prompt the user to save the project and thus the changes are lost. Another thing to do is making error messages from the parsing process to be displayed in the GUI:s text window, instead of the shell.

It is, however, not always so easy to locate a bug or the reason for it in the first place, and thus it can be a quite time-consuming process. This is why these have been left as future work.

Chapter 7

Acknowledgements

I wish to extend a great thanks to my supervisor Robert Granat for providing me with the opportunity to undertake this Master's Thesis project and for all the help he has provided.

I would also like to thank the previous designers of OpenFGG for the help which I have received, especially Magnus Andersson whose knowledge about OpenFGG's core functionalities and Fortran parsing has been an invaluable asset to this project.

References

[1] Magnus Andersson. Semi-Automatic Generation of MATLAB Gateway Functions of Numerical Fortran Routines. *Master's Thesis*, Report UMNAD 561/05, Dept. Computing Science, Umeå Univesity, 2005.

[2] Johan Sejdhage. Designing and Implementing a Java-based GUI for Automatic Generation of MATLAB Gateway Functions of Numerical Subroutines. *Master's Thesis*, Report UMNAD 572/04, Dept. Computing Science, Umeå Univesity, 2005.

[3] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, fifth edition, 2002.

[4] Michael Metcalf, John Reid. *Fortran 90/95 Explained*. Oxford University Press, second edition, 2000.

[5] Brian Meek. *The Fortran (not the foresight) saga: the light and the dark*. The Fortran Company. Web page, updated 22 October 1996. `http://www.fortran.com/forsaga.html`

[6] Bo Einarsson. *Lärobok i Fortran 90/95*. Linköping University, 2004.

[7] Compaq Fortran. *Language Reference Manual*. Digital Equipment Corporation, 1999.

[8] Inc. Sun Microsystems. Java^{TM}2 Platform Standard Edition 5.0 API Specification. Web page, 23 May 2006. `http://java.sun.com/j2se/1.5.0/docs/api/`

Appendix A

Notes on Using OpenFGG 1.5

There are some notes one should read before using the new release of OpenFGG, version 1.5.

Using the application is the same as for version 1.0 except for the following:

1. Old project files does not work with this version of OpenFGG.
2. In the *Project setup* window there is a new component below the *Source files* window called *Source form*. It consists of two radio buttons which indicate if OpenFGG should treat the selected source-file as written in fix or free source form. By default, the source form is determined by the extension of the file, but it can also be set manually by clicking on the corresponding radio button.

   See figure A.1 for an example.
3. The mode of a parameter which makes use of the `INTENT` attribute is automatically filled in after parsing.
4. Using *assumed-size* arrays as arguments no longer generate the "DIMENSION REQUIRED" message in the expression table of the GUI.
5. For simplicity and efficiency, only source code that contains declarations will be parsed in detail. A consequence of this is that dummy procedures now must be declared with the `EXTERNAL` keyword in order to be detected.

For general information on how to use OpenFGG, see the User's Guide which can be found at OpenFGG's homepage[1].

[1] http://www.cs.umu.se/research/parallel/openfgg/

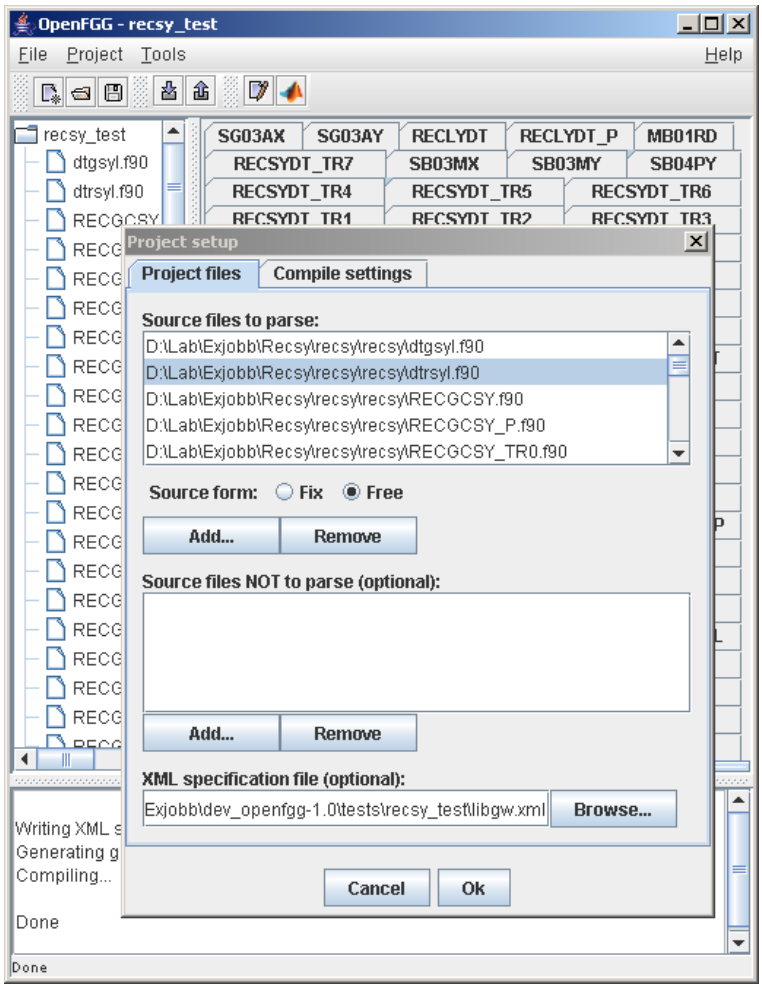

Figure A.1: Selecting source form in OpenFGG.

Appendix B

XML Document Format

Below is the document type definition (DTD) of the XML-file which is used in OpenFGG projects to store information about the gateway. It has been originally defined by Magnus Andersson and the only change made to it is the addition of the `optional` attribute for the elements `variable`, `array` and `split-complex`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!ENTITY % mode "input|output|inout|work">
<!ENTITY % ncntype
    "INTEGER|REAL|DOUBLEPRECISION|integer|real|doubleprecision"
>
<!ENTITY % cntype
    "COMPLEX|DOUBLECOMPLEX|complex|doublecomplex"
>
<!ENTITY % ntype "%ncntype;|%cntype;">
<!ENTITY % nchtype "%ntype;|LOGICAL|logical">
<!ENTITY % type "%nchtype;|CHARACTER|character">

<!ELEMENT gateway (options?, source+)>
<!ATTLIST gateway
    exename NMTOKEN #REQUIRED
    srcname NMTOKEN #REQUIRED
>

<!ELEMENT options (xerbla?)>

<!ELEMENT xerbla EMPTY>

<!ELEMENT source (function|subroutine)+>
<!ATTLIST source
    file CDATA #REQUIRED
>

<!ELEMENT subroutine
    (description?, (variable|array|split-complex|returnspec)*)
>
```

```
<!ATTLIST subroutine
    name CDATA #REQUIRED
>

<!ELEMENT function
    (description?, (variable|array|split-complex)*)
>
<!--
len and bound attributes are only used when type is CHARACTER.
If left out 1 is implied.
-->
<!ATTLIST function
    name CDATA #REQUIRED
    type (%type;) #REQUIRED
    len CDATA #IMPLIED
    bound CDATA #IMPLIED
    mode (output|work) #REQUIRED
>

<!ELEMENT description (#PCDATA)>

<!ELEMENT returnspec EMPTY>

<!ELEMENT variable (init?)>
<!--
len and bound attributes are only used when type is CHARACTER.
If left out 1 is implied.
-->
<!ATTLIST variable
    type (%type;) #REQUIRED
    len CDATA #IMPLIED
    bound CDATA #IMPLIED
    name CDATA #REQUIRED
    optional CDATA #IMPLIED
    matlabname CDATA #IMPLIED
    mode (%mode;) #REQUIRED
>

<!ELEMENT init (#PCDATA)>

<!ELEMENT array (dim+)>
<!ATTLIST array
    type (%nchtype;) #REQUIRED
    name CDATA #REQUIRED
    optional CDATA #IMPLIED
    mode (%mode;) #REQUIRED
>
<!ELEMENT dim (#PCDATA)>
```

```
<!ELEMENT split-complex (((real,imaginary)|(imaginary,real)),
                         (dim*))>
<!ATTLIST split-complex
    type (%ncntype;) #REQUIRED
    name CDATA #IMPLIED
    optional CDATA #IMPLIED
    mode (%mode;) #REQUIRED
>

<!ELEMENT real EMPTY>
<!ATTLIST real
    name CDATA #REQUIRED
>

<!ELEMENT imaginary EMPTY>
<!ATTLIST imaginary
    name CDATA #REQUIRED
>
```