User's Guide to f2j
Version 0.8

Keith Seymour and Jack Dongarra

Innovative Computing Laboratory
Department of Computer Science
University of Tennessee

May 31, 2007

1 Introduction

Before using the f2j source code, realize that f2j was originally geared to a very specific problem - that is, translating the LAPACK and BLAS numerical libraries. However, now that the translation of the single and double precision versions of BLAS and LAPACK is complete, the goal is to handle as much Fortran as possible, but there's still a lot left to cover. We have a lot of confidence in the JLAPACK translation, but for a variety of reasons, f2j will most likely not correctly translate your code at first.

One of the reasons for putting the code up on SourceForge is to enable easy collaboration with other developers. If you're interested in helping the development of f2j, we'll consider giving commit access to the CVS tree.

The purpose of this document is to describe how to build and use the f2j compiler and to give some background on how to extend it to handle your Fortran code.

2 Obtaining the Code

For downloads and CVS access, see the f2j project page at SourceForge:

```
http://sourceforge.net/projects/f2j
```

There is a source tarball available in the download section and anonymous CVS access is also available.

The GOTO translation code is based on the bytecode parser found in javab, a bytecode parallelizing tool under development at the University of Indiana. That code is covered under its original license, found in the translator source directory.

3 Limitations

There are many limitations to be aware of before using f2j:

- Parsing – the parser has a bug that requires at least one variable declaration in every program unit. Also, the last line of the program cannot be blank.
- Typechecking – f2j does not aim to do much typechecking. It assumes that you have already tested the code with a real Fortran compiler.
- Data types – complex numbers are not supported.
- Input/Output – f2j does not support any file I/O. Formatted I/O support is fairly weak, but works for many simple cases. At worst, the output will be missing or you'll get "NULL" printed out instead of numbers.
- Other Features – Certain forms of Fortran EQUIVALENCE are not supported. f2j can handle a limited form of EQUIVALENCE as long as the variables being equivalenced do not differ in type and are not offset from each other. Multiple entry points are not supported.

With that said, if you have pretty straightforward numerical code (similar to BLAS or LAPACK) f2j may be able to handle it.

4 Building and Using f2java

We have been doing development and testing of f2j on Sun SPARCstations running various versions of Solaris as well as x86 machines running various versions of Linux and Solaris/x86. It may compile on other platforms, though. Using gcc 3.4.4 with the `-Wall` flag, we get no warnings, but using some picky compilers, you may see warnings about unused variables, etc. You can safely ignore them.

First, download and uncompress the source code. Building the code follows the typical configure/make process:

```
# ./configure
# make
```

Optionally, you can "make install" which will copy the executables to the location specified in the `--prefix` argument to configure.

Now you may want to add the relevant install directory to your PATH. This will vary depending on whether you did "make install". If so, the PATH should include `$prefix/bin`. If not, your PATH should include `$f2j_dir/src` and `$f2j_dir/goto_trans`, where `$f2j_dir` is the top-level f2j source directory. You may also want to modify your CLASSPATH to include the f2j util package. If you did "make install", this will be `$prefix/lib/f2jutil.jar`. Otherwise, it will be `$f2j_dir/util/f2jutil.jar`.

Let's go through a simple example. Say you have the following Fortran code in a file called "test.f"

```
      program blah
      external foo
      write(*,*) 'hi'
      call foo(12)
      stop
      end
      subroutine foo(y)
      integer y
      write(*,*) 'foo ', y
      return
      end
```

If you translate it with "`f2java test.f`", it will produce one class file and one Java source file for each program unit. So, in this case since we have two program units in the Fortran source file, we end up with four generated files: Blah.java, Blah.class, Foo.java, and Foo.class (note the first letter of the name becomes capitalized). You can run the generated class file directly:

```
# java Blah
hi
foo  12
```

You don't need to compile the Java source, but if you wanted to modify it, you could recompile:

```
# javac Blah.java Foo.java
```

However at this point the GOTO statements haven't been converted, so if you run it you'll see some warnings like this:

```
# java Blah
hi
foo  12
Warning: Untransformed goto remaining in program! (Foo, 999999)
Warning: Untransformed label remaining in program! (Foo, 999999)
```

So you need to run the GOTO transformer (javab) on the class files:

```
# javab *.class
```

and then it'll run fine:

```
# java Blah
hi
foo  12
```

5 Command-line Options

There are several command-line options that you should be aware of:

- -I specifies a path to be searched for included files (may be used multiple times).
- -c specifies the search path for f2j "descriptor" files (ending in .f2j). It is a colon-separated list of paths, like a Java CLASSPATH). For example:

  ```
  f2java -c .:../objects filename.f
  ```

- -p specifies the name of the package. For example:

  ```
  f2java -p org.netlib.blas filename.f
  ```

- -o specifies the destination directory to which the code should be written.
- -w forces all scalars to be generated as wrapped objects. The default behavior is to only wrap those scalars that must be passed by reference. Note that using this option will generate less efficient Java code.
- -i causes f2j to generate a high-level interface to each subroutine and function. The high-level interface uses a Java-style calling convention (2D row-major arrays, etc). The low-level routine is still generated because the high-level interface simply performs some conversions and then calls the low-level routine.
- -h displays help information.
- -s causes f2j to simplify the interfaces by removing the offset parameter and using a zero offset. It isn't necessary to specify -i in addition to -s.

- -d causes f2j to generate comments in a format suitable for javadoc. It is a bit of a LAPACK-specific hack – the longest comment in the program unit is placed in the javadoc comment. It works fine for BLAS/LAPACK code (or any other code where the longest comment is the one that describes the function), but will most likely not work for other code.
- -fm causes f2j to generate code that calls java.lang.StrictMath instead of java.lang.Math. By default, java.lang.Math is used.
- -fs causes f2j to declare the generated code as strictfp (strict floating point). By default, the generated code is not strict.
- -fb enables both the -fm and -fs options.
- -vs causes f2j to generate all variables as static class variables. By default f2j generates variables as locals.
- -va causes f2j to generate arrays as static class variables, but other variables are generated as locals.

After issuing the command "f2java file.f" there should be one or more Java files in your current directory, one Java file and one class file per Fortran program unit (function, subroutine, program) in the source file. Initially, we would suggest concatenating all Fortran program units into one file because it makes it easier to perform correct code generation (more about this later). As the example above illustrated, you can run the class file corresponding to the main Fortran program unit or you can use the Java compiler of your choice to compile the resulting Java source code. Make sure that the org.netlib.util package is in your CLASSPATH. This package comes in both the f2j and JLAPACK distributions, so if your CLASSPATH already points to JLAPACK's f2jutil.jar, then you're ok.

6 Organizing Your Fortran Code

Any non-trivial Fortran program will consist of multiple source files, often in many different directories. This can present difficulties for f2j because resolving external functions and subroutines is critical for generating the call correctly.

First, we will give some practical advice on organizing your code to be built using f2j. The following section will give a more detailed explanation of why this is all so important.

6.1 Practical Aspects

The easiest method is to just concatenate all your Fortran code into one file and run f2j on it. This might not be practical in all cases, though. If you have to keep code in separate files, you need to understand the dependence relationship between them. For example, if you have files `a.f` and `b.f`, and routines in `a.f` call routines in `b.f`, then you must translate `b.f` first. If there is a cross dependency, then f2j will most likely not generate some calls correctly. Thinking of it as a call tree, you want to start translating at the leaves and work your way back up. This sometimes requires modifying the code.

When code exists in separate subdirectories, the procedure is largely the same, except that f2j needs to know the subdirectory names containing files that the current program unit depends on.

Calling FUNC1	Calling FUNC2
`getstatic #15 <Field Hello.x:int[]>`	`getstatic #15 <Field Hello.x:int[]>`
`iconst_5`	`iconst_5`
`iconst_1`	`iconst_1`
`isub`	`isub`
`iaload`	`invokestatic #28`
`invokestatic #22`	`<Method Func2.func2(int[],int):void>`
`<Method Func1.func1(int):void>`	

Table 1: Differences in Argument Passing.

Modifying the previous example, let's say that `b.f` is in a subdirectory named `../code/foo`. We would first go to `../code/foo` and translate `b.f`, which would result in the creation of a number of descriptor files ending in `.f2j`. Then in the subdirectory containing `a.f`, specify the other subdirectory on the command line:

```
# f2java -c .:../code/foo a.f
```

f2j will locate the descriptor files in `../code/foo` and use them to generate the correct calls to the routines contained in `b.f`. You can specify multiple paths separated by a colon.

6.2 Resolving External Routines

This section illustrates in more detail the importance of resolving calls to functions or subroutines which do not appear in the original source file. By "resolving", we mean determining the correct calling sequence for the function call, which depends on its method signature. For example, consider the following Fortran program segment:

```
      INTEGER X(10)

      CALL FUNC1( X(5) )
      CALL FUNC2( X(5) )
[...]
      SUBROUTINE FUNC1(A)
      INTEGER A
[...]
      SUBROUTINE FUNC2(A)
      INTEGER A(*)
```

The first subroutine, `FUNC1`, expects a scalar argument, while `FUNC2` expects an array argument. These two calls would be generated identically in a standard Fortran compiler, regardless of how `FUNC1` and `FUNC2` were defined — the address of the fifth element of X would be passed to the subroutine in both cases. However, things are not as simple in Java due to the lack of pointers. To simulate passing array subsections, as necessary for the second call, we actually pass two arguments — the array reference and an additional integer offset parameter, as shown in the right column of Table **??**.

However, the first subroutine expects a scalar, so we should pass only the value of the fifth element, without any offset parameter, as shown in the left column of Table **??** (in this case, assume that `FUNC1` does not modify the argument, otherwise things get even more complex).

Notice that the primary difference between the two calling sequences is that when calling `FUNC1`, the array is first dereferenced using the `iaload` instruction. Also note that the purpose of the arithmetic expression is to decrement the index by 1 to compensate for the fact that Java has 0-based indexing whereas Fortran has 1-based indexing.

The only way to determine the correct calling sequence for any given call is to examine the parameters of the corresponding subroutine or function declaration. This is only possible if the declaration had been parsed at the same time as the current program unit, meaning that for code generation to work properly all the source files had to be joined into a big monolithic input file.

This was a serious limitation, especially for large libraries, because a modification to any part of the code requires re-compiling *all* the source. There are at least a couple of ways to solve this problem. One way would be to obtain the parameter information directly from class files that have already been generated. While this would work well, f2j is written in C and does not have access to nice Java features like reflection, so it would require a lot of extra code to parse the class files. Instead, we use a more lightweight procedure in f2j. At compile-time, f2j creates a *descriptor file* which is a text file containing a list of every method generated. Each line of the descriptor file contains the following information:

- Class name – the fully qualified class name which contains the given method.
- Method name – the name of the method itself.
- Method descriptor – this method's descriptor, which is a string representing the types of all the arguments as well as the return type.

Continuing with the previous example, the descriptor files for `FUNC1` and `FUNC2` would be:

```
# cat Func1.f2j
Func1:func1:(I)V

# cat Func2.f2j
Func2:func2:([II)V
```

To resolve a subroutine or function call, we search all the descriptor files for the matching method name and examine the method descriptor. Based on the method descriptor, we can then correctly generate the calling sequence. The code generator locates the descriptor files based on colon-separated paths specified on the command line or in the environment variable `F2J_SEARCH_PATH`.

7 Extending f2j

So, at this point you may be wondering how to extend f2j to handle your code. Typically, the first problem you'll run into is that f2j doesn't parse your code. That could involve something as simple as changing a production in the parser or it could involve a bit more work - e.g. creating a new kind of AST node along with all the appropriate code generation routines. The first thing you'll want to check is whether the parser supports the syntax your code uses (the parsing code is machine generated from a Yacc grammar in f2jparse.y). For example, if your code contains an ENTRY

statement, your code will not compile because f2j doesn't support alternate entry points. Suppose you wanted to implement ENTRY in f2j. Your first step would be to define a lexer token to represent the ENTRY keyword (in fact, this exists already, even though ENTRY is not implemented). The lexer sometimes needs to be modified to handle the token correctly, but usually it is sufficient to put the token in the appropriate lexer table. In this case, we would just put the ENTRY keyword in the `tab_stmt` array defined in `globals.c`. That array holds keywords that are at the beginning of statements. You'll notice that this has also been added already.

If you're getting parse errors on a line of code that should compile based on your examination of the parser, then the lexer might not be sending the correct tokens to the parser. The lexical analysis code is in f2jlex.c, which is handwritten C code based on Sale's algorithm. There's not really an easy way of describing the structure of the lexer code, but if you enable debugging output (set lexdebug = TRUE) it will show which tokens are being passed from the lexer to the parser. That should help you figure out where the problem is.

While you're working on the parsing, you can leave the code section in the Yacc grammar blank. You'll recognize when it finally parses correctly because you'll get a segmentation fault (meaning it passed the parsing phase and failed in a subsequent phase since you didn't pass an AST node back up from that production). At this point, you need to determine what information is needed by the back-end to generate the code. For example, a loop might need a statement label number, an initial value, a final value, and an increment value. The AST node types are defined in `f2j.h`. If the node you're defining is close enough to an existing node, you can reuse it. Otherwise you'll have to create a new one. Then just initialize this node in the code section for your new production.

If f2j can parse your code, but the resulting Java code does not compile or does not work, then this may indicate a problem in the f2j back-end. First, try concatenating all your Fortran files into one big file (ok, we admit this is cheesy, but it does work sometimes). This should help with the type analysis phase and may eliminate problems in the resulting Java code. After that, if the generated code is still incorrect, begin looking into the f2j code. After f2j parses your code, it passes through a couple of stages before actually generating code. First, the AST goes through "type analysis" (typecheck.c), which simply means that the tree is fully traversed and each node is assigned type information as appropriate. This is not semantic analysis, just annotation. Next, the AST goes through "scalar optimization" (optimize.c), which is an optimization stage designed to determine which scalar variables need to be wrapped in objects and which can remain primitives. After that, f2j generates the Java code (codegen.c) based on the modified AST. So, if you notice a type mismatch problem in the generated code, typecheck.c would be a good place to begin debugging. Similarly, if you notice that object wrappers are inappropriately used, check into optimize.c (hint: by passing the -w flag to f2java, the scalar optimization code will be skipped). Most other problems will be with the code generator itself.