JLAPACK—Compiling LAPACK FORTRAN to Java, Phase 1
Technical Report cs-97-367

David M. Doolin* Jack Dongarra* †

August 21, 1997

Abstract

The JLAPACK project will provide the LAPACK numerical subroutines translated from their subset FORTRAN 77 source into class files, executable by the Java Virtual Machine (JVM) and suitable for use by Java programmers. This makes it possible for a Java application or applet, distributed on the World Wide Web (WWW) to use established legacy numerical code that was originally written in FORTRAN. The translation is accomplished using a special purpose FORTRAN-to-Java compiler. This interim report describes the research issues involved in the JLAPACK project, and its current implementation and status.

1 Introduction

Real programmers program in FORTRAN, *and can do so in any language. —Ian Graham, 1994 [1]*

Popular opinion seems to hold the somewhat erroneous view that Java is "too slow" for numerical programming. However, the Java Linpack benchmark (`http://www.netlib.org/benchmark/linpackjava/`) has recorded excellent floating point arithmetic speeds (43.5 Mflop) on a PC resulting from Just-In-Time (JIT) compilation of Java class files. Also, there are many small to intermediate scale problems where speed is not an issue. For instance, physical quantities such as permeability, stress and strain are commonly represented by ellipsoids [2, 3], a graphical representation of an underlying tensor. The tensor is mathematically represented by an SPD matrix. Ellipsoid axes are computed from the root inverse of the matrix eigenvalues, directed along the eigenvectors. A LAPACK eigenproblem subroutine such as SSYTRD, available as a Java class file, provides a portable solution with known reliability. Since future execution speeds of Java will increase as JIT and native code compilers are developed, the scale of feasible numerical programming will increase as well.

The JLAPACK project will provide Application Programming Interfaces (APIs) to numerical libraries from Java programs. The numerical libraries will be distibuted as class files produced by a FORTRAN-to-Java translator, f2j. The f2j translator is a formal compiler that translates programs written using a subset of FORTRAN 77 into a form that may be compiled or assembled into Java class files. The first priority for f2j is to translate the BLAS [4, 5, 6] and LAPACK [7] numerical libraries from their FORTRAN 77 reference source code to Java class files. The subset of FORTRAN 77 translated by f2j matches the Fortran source used by BLAS and LAPACK. These libraries are established, reliable and widely used linear algebra packages, and are therefore a reasonable first testbed for f2j. Many other libraries of interest are expected to use a very similar subset of FORTRAN 77.

A similar previous translation effort provided LAPACK in the C language, using the f2c program [8], and has proven to be very popular and widely used. The BLAS and LAPACK class files

*Department of Computer Science, University of Tennessee, TN 37996

†Mathematical Science Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831

will be provided as a service of the Netlib repository. f2j also provides a base for a more ambitious effort translating a larger subset of Fortran, and perhaps eventually *any* Fortran source into Java class files.

The JLAPACK project is composed of three phases:

1. Phase 1: Writing a FORTRAN compiler front end to tokenize (lexically analyze), parse and construct an abstract syntax tree (AST) for FORTRAN 77 input files.

2. Phase 2: Generate Java source and Jasmin opcode for use with JVM from the AST.

3. Phase 3: Test, document and distribute BLAS and LAPACK class files.

Phase 1 of the JLAPACK project is complete with respect to the initial design criteria. Significant progress has been made in Phase 2: Translation to Java source is complete within the initial design; translation to Jasmin opcode is roughly 50% complete. Implementing Phase 3 will require extending the FORTRAN front end to handle initially unimplemented parts of FORTRAN, which is necessary to translate certain LAPACK files. These extensions, along with implementing simple input/output, should also allow translation of the BLAS and LAPACK test suites.

2 Design of the f2j compiler

u

Design issues come in two categories: (1) software design, (2) software implementation. Software design specifies how FORTRAN translates to Java independent of any implementation. This includes dealing with general issues such as translating FORTRAN intrinsics (e.g., `sqrt, dabs`) to Java methods (e.g., `Math.sqrt, Math.abs`), and LAPACK specific decisions about array access and argument passing. The software implementation executes the translation. f2j is written as a formal compiler consisting of a lexical analysis and parser front end, and a code generation back end. The code is written in C, and C is produced by the parser generator, bison, used to specify the FORTRAN grammar and create the abstract syntax tree (AST) data structure. The following notes provide a general overview of the f2j software design and implementation.

2.1 Translating LAPACK FORTRAN to Java

2.1.1 Argument passing

Parameters passed to the LAPACK driver routines consist of arrays, floating point numbers, integers, and arrays of one or more characters. Arrays are objects in Java and are passed by reference similar to how they are passed in FORTRAN. Character arrays are similar to String objects in Java, which are passed by reference (details in §2.1.4). Primitive types such as integers and floats are passed by value only in Java and by reference only in FORTRAN. Since objects require more overhead than primitives, the number of primitives passed as objects should be minimized.

All primitives that are documented as "input/output" or "output" variables in the LAPACK code can be handled by wrapping the value in a class definition and instantiating an object for initializing the value. A simple experiment showed that instantiating an object of type Double requires 280 bytes in Java (Sun Microsystems jdk-1.1), but a simple wrapper such as

```
class DoubleWrapper {
  double d;
}
```

only requires 56 bytes. Using the appropriate object variables (input/output and output variables) should not be an excessive burden on the user: programmers calling LAPACK from C must declare, initialize and pass a pointer to these variable.

2.1.2 Array access

Arrays in Java differ from arrays in FORTRAN in several ways. In Java, arrays are objects that contain methods as well as data, thus increasing overhead. In FORTRAN, arrays are named contiguous blocks of memory. Java allows arrays as large 255 dimensions; FORTRAN allows a maximum of 3 dimensions. Array indices must start at 0 in Java but can start at arbitrary integer, say -43, in FORTRAN. Java is implemented as row major access, FORTRAN as column access. FORTRAN also allows sections of arrays to be passed as subarrays.

For instance, in FORTRAN a reference to an arbitrary point in an array may be passed to a subroutine. A call such as `matmult( A(i,j), B(i,j) )` would pass in the arrays `A` and `B` to the `matmult` procedure, which would start indexing the arrays `A` and `B` at the location `i, j`. Java dereferences and passes the value in the array at position `i, j`. Similarly, one can pass in a single reference that marks a location in a particular array, which is declared 2D when typed in the called subroutine. These and similar conventions allow numerical analysts to construct efficient algorithms.

In the JLAPACK subroutines, all arrays are declared 1D. For JLAPACK vectors, array access is identical to FORTRAN. Since the vector may be accessed at a point other than the initial point, an index is passed along with the array. For 2D arrays, the index is passed as parameter indicating an offset from the $0th$ element. The leading dimension is also passed as a parameter. To enable future optimization by minimizing index arithmetic, arrays are accessed in column order in JLAPACK.

For example, a FORTRAN call such as `matrixop(A(i,j,), LDA)` would be translated to Java as `matrixop(A, i+j*LDA, LDA)`, where `i, j` are the array indices, and `LDA` is the previously declared leading dimension. The `matrixop` method would recieve arguments thusly: `(double [] A, int k, int LDA)`, `k` indicating the offset. Elements in the subarray starting at the location `i, j` would accessed by `A[k + m + n*LDA]`, where `m, n` are loop counters. In column order access, part of the index arithmetic could be moved outside the inner loop, reducing the number of operations per iteration.

Three timing loops (Appendix B) written to compare the execution speed of of 1D versus 2D arrays returned mean speeds ($n = 32$) of 482 for 2D arrays, 592 for 1D row access arrays and 462 for 1D column access arrays. The column access array moved an index product term to a dummy variable between the outer and inner loops. Single dimension arrays also provide an easy way to deal with assumed-size array declarators (asterisks) in FORTRAN. Subroutine and function arguments in FORTRAN must be typed after the arguments are declared, as the following code illustrates:

```
SUBROUTINE DLASSQ( N, X, INCX, SCALE, SUMSQ )
...
DOUBLE PRECISION   X( * )
```

But DLASSQ is called from DLANSB with the 2D array `AB`:

```
  CALL DLASSQ( N, AB( L, 1 ), LDAB, SCALE, SUM )
```

Since there is no similar syntax in Java, 1D arrays provide equivalent functionality.

2.1.3 Translating functions and subroutines

Translating functions and subroutines from FORTRAN to Java can be broken down into various cases:

- Subroutines and functions declared EXTERNAL are assumed, for the purpose of translating BLAS and LAPACK, to be BLAS or LAPACK calls. These are translated by the `call_emit` procedure during the code generation pass of f2j. Note that these are tailored to LAPACK: generated code assumes one static method per class.
- Some functions and subroutines in BLAS and LAPACK correspond to methods intrinsic to Java. The LSAME procedure, which compares characters indpendent of case, is an example corresponding to the Java `ignoreEqualsCase` method.

- Some subroutines were hand translated because they were very small, or used unimplemented keywords such as those used for I/O. `xerbla`, an error handling routine, is an example.
- Functions declared INTRINSIC in the BLAS and LAPACK FORTRAN source are mapped to the corresponding Java method using a table initialized in a header file. In the event that a FORTRAN intrinsic procedure has no Java correspondence (e.g., complex arithmetic operations), such methods will have to be programmed in Java.

2.1.4 Translating characters and strings

LAPACK uses alphabetic characters and as flags to control the behavior of called subroutines, and character arrays to print out diagnostic information such as which subroutines or functions encountered an error. Java uses String objects instead of character arrays. For the purpose of translating LAPACK into Java, all FORTRAN character variables, whether single characters or character arrays, are translated to Java String objects. Subroutines in LAPACK, such as LSAME which compares characters independent of case, can be emulated with methods intrinsic to the native Java String class.

2.1.5 The PARAMETER keyword

The PARAMETER declaration in FORTRAN is translated to a `public static final` declaration in java.

2.1.6 Variable inititialization

One problem that has cropped up for emitting Java source code is the `INCX` problem. `INCX` is passed in as a parameter to certain routines and used to set the values of variables `KX`. The problem occurs after the following test:

```
if (INCX <= 0)
  {
      KX = 1 - (N - 1) * INCX;
  }                    // Close if()

else if (INCX != 1)
  {
      KX = 1;
  }                    // Close else if()
```

`KX` is not initialized, and is not required to be initialized according to the F77 specification. The problem occurs when `INCX = 1`. The Java compiler refuses to compile any code following this that uses `KX`. A solution is implemented in the compiler by setting the value of `KX` to zero. Since variable initialization is implementation dependent in FORTRAN, the f2j implementation initializes all integer variables to zero.

2.1.7 Methods versus functions and subroutines

Call_emit is a BLAS/LAPACK specific procedure. The method takes a name that is not in any table, and assumes that it refers to a method available in the netlib packages. For instance, the function ddot is translated to `Ddot.ddot()`, that is, the method `ddot` of the class `Ddot`. This is not a portable solution and will break if the called function is *not* in the BLAS or LAPACK packages.

In FORTRAN functions, the function name may be initialized to a value as a variable would be, then the function name is the implicit argument when `return` is called. FORTRAN functions return a typed (double, etc.) value. Subroutines are analogous to void. Both are handled by methods in Java.

2.2 Implemention of the f2j compiler

The program f2c is a horror, based on ancient code and hacked unmercifully. Users are only supposed to look at its C output, not at its appalling inner workings. —Stuart Feldman [8]

The f2j compiler system was written in ANSI C, using a C parser generated by the Bison parser generator. The code was written from scratch after determining that existing FORTRAN tools such as f2c and g77 would be difficult to modify. Similarly, the Bison grammar was derived from the FORTRAN 77 standard since available parse files would have needed extensive rewriting to produce an abstract syntax tree (AST). The BLAS and LAPACK source code are assumed to be syntactically correct FORTRAN. Comments in the compiler source are written as complete sentences, starting with a capital letter and ending with a period. Comments from the FORTRAN source are not preserved in the translation, though this is recommended as a desirable if not necessary extension. Separate executables are compiled to generate Java source code (f2java) and Jasmin opcode (f2jas).

2.2.1 Lexing FORTRAN

It should be noted that tokenizing FORTRAN *is such an irregular task that it is frequently easier to write an ad hoc lexical analyzer for* FORTRAN *in a conventional programming language than it is to use an automatic lexical analyzer generator. —Alfred Aho, 1988 [9]*

Lexing FORTRAN is somewhat difficult because keywords (e.g., IF, DO, etc.) are not reserved. Thus keywords can also be used variable names. To properly lex FORTRAN, each statement must be examined for context, which requires lookahead. Sale published an algorithm for lexing FORTRAN in CACM in the 1960's. Fortunately, once FORTRAN is lexed, it is fairly easy to parse.

In general, the f2j lexer aspires to be FORTRAN specific variant of the *lex* lexical analysis tool. The compiler uses global variables so that the parser's `yyparse` procedure can communicate with the lexer's `yylex` procedure. Global variables, such as `yytext` and `yylval`, are typed in the header file f2j.h, and declared `extern` in the functions that use them. Line numbers are counted, but the information is not currently used in the compiler.

Lexing in f2j is a two phase process consisting of a scan phase and a lexical analysis phase. The scan phase removes whitespace and comments, catenates continued lines together, marks the end of file and implements Sale's algorithm (Appendix A) to determine context. The lexical analysis phase implements a custom-written `yylex` procedure to provide tokens to the parser, `yyparse`, which is generated by the *Bison* parser generator. `yylex` implements a scan phase at the beginning of every FORTRAN statement by calling several procedures to manipulate the statement input string from the source file into a valid FORTRAN statement. The statement is scanned, lexed, and the next token, along with its lexical value, is made available to the parser.

At the beginning of every FORTRAN input statement, `prelex()` is called to read a line from the FORTRAN input file, disposing of comment lines until a valid line is found. Once a valid line is found, the line buffer is passed to `check_continued_lines()`, which does a look-ahead to the sixth column of the next line. If there is a '$' character in the sixth position, the next line is read and catenated to the previous line, incrementing the (global) file pointer. If there is no $, the file pointer is reset for the next call to `prelex()`.

Once `check_continued_lines()` returns a complete statement, `prelex()` calls `collapse_white_space()` to remove all spaces, tabs and newlines from the statement. Extra newlines embedded between continued lines would result in a parse error since newlines are used as FORTRAN's statement delimiter. This is done in a loop, incrementing a character pointer that is dereferenced to compare characters. After all whitespace is removed, one newline is catenated to the very end of the statement which can be passed as a token to the parser. `collapse_white_space()` also changes characters into upper case, with the exception of FORTRAN character arrays, enclosed between tick (') marks, that are passed back as literal text in the statement buffer as well as in the text buffer. Also, Sale's algorithm

(Appendix A) is implemented to determine context. Once the white space is removed, `prelex()` increments the line number and control passes back to `yylex()`.

In the lexical analysis phase, statements are scanned for tokens according to the context determined from the prepass. The `yylex()` procedure calls one of three scanning procedures, `keyscan()`, `name_scan()`, or `number_scan()` to extract tokens from statements. `keyscan()` takes tables of keywords or symbols defined as part of the FORTRAN language language, along with the statement buffers returned from the prepass. `name_scan` and `number_scan()` only take the statement buffer arguments. All scanning routines modify the statement buffers and return tokens along with any lexical values present.

The `keyscan()` routine takes one of three tables defined in an intialization header file. The tables contain either keywords, types or symbols. The appropriate table is chosen by context determined from the prepass. Table scanning is accompilshed by determining the length of the word or symbol string, then string comparing to determine a match. A successful match advances a character pointer to the end of the new token, which is returned along with any lexical value. The remaining string is copied into the statement buffer. The lexical values are determined from the matching source code text buffer. The tables are split into three types: one for FORTRAN key words (IF, DO, etc.), one for FORTRAN types (REAL, INTEGER, etc.), and one for symbols (+, =, etc.).

The `name_scan()` if there is no context for a key word, and if the character pointer to by the statement buffer is alphabetic. `name_scan()` loops over the characters in the statement buffer, advancing a character pointer until a non-alphanumeric character is seen. Then the statement and text buffers are updated and the NAME token is passed back to `yylex()`. The lexical value is copied into the global union variable `yylval` for use by `yyparse()` when NAME is reduced.

The `number_scan()` procedure addresses some other lexical questions associated with FORTRAN, such as the look-ahead needed to determine whether the characters "123" reduce to an integer in the relational operation `123.EQ.I`. This is accomplished by advancing a character pointer over the statement buffer while the current character is a digit or any of the characters in the set {D, d, E, e, .}. Encountering a "." during the loop causes the required look-ahead to determine whether the number is an integer or a floating point number. Encountering any of the letter D, d, E, or e invokes more look-ahead to determine the sign of the associated exponent. The procedure returns tokens and values similarly to `name_scan()`.

2.2.2 Parsing FORTRAN

> *One may reasonably ask why anyone would use* FORTRAN *today, since experts seem to agree that the language is obsolete—Stuart Feldman, 1979 [10]*

The FORTRAN grammar has been described as neither LL or LR, or LL and not LR, or LR and not LL, or both LL and LR. All answers are partly correct. FORTRAN was written before the notion of regular expressions, and before context-free grammars were derived. FORTRAN predates Knuth's (get ref) derivation of LR parsing by about 10 years. Fortunately, the LAPACK subset of FORTRAN 77 may be parsed LR(1), once the lexical structure has been determined.

The yacc grammar was written using the FORTRAN77 standard [11]. The grammar was implemented using the `Bison` parser generator, a yacc work-alike distributed by the Free Software Foundation. Bison generates an ANSI C parser, which helps ensure platform independence. The Fortran source code is parsed into an abstract syntax tree (AST) consisting of tagged union nodes implementing the equivalent Java structures. The AST allows easy lookup and connection between non-adjacent nodes if future code restructuring is desired. The AST can be passed by its root node to separate type-checking, code optimizing and code generation procedures.

The compiler uses global variables to communicate between the lexer and the parser because the Bison generated parser routine `yyparse()` is automatically generated and takes no arguments. Gobal variables are declared in the `f2j.h` header file and as `extern` in the functions that use them. In the parser, tokens that are associated with a lexical value are immediately reduced to store the value. Since Bison reduces on in-line actions, in-line actions are avoided when possible.

The abstract syntax tree (AST) contains a single node type consisting of an enum statement naming FORTRAN constructions (e.g., do loop), a union of structs to store data for each type of FORTRAN construction, pointers to attach the nodes, and a token value. Some of the structs in the union, such as that used for assignments, can also be used for expressions during the parsing pass. The node would be assigned the appropriate tag (enum variable) because code generation procedures are necessarily different for assignment statements and expressions. The pointers are used to doubly link statement blocks, or in the case of expressions, parent from child nodes to parent nodes. Since Bison is an LR parser, linked list are built in reverse, and must be switched for inorder traversal. The switching is done before sublists are attached to the main part of the program, and for the main program, directly after the final grammar reduction before the code generation routines are invoked.

The AST contains data for an entire FORTRAN program unit specified by the FORTRAN keywords `program`, `function` or `subroutine`. Each FORTRAN program unit consists of statement blocks, which are composed of one or more FORTRAN statements. The statement blocks are connected into a doubly linked list, reflecting FORTRAN's procedural control of flow.

The statement block may consist of a single statement, or a group of statements in a control structure such as a do-loop or if-then-else block. Statements within the scope of such structures are linked as a sublist within the block instead of sequentially along the main flow of the program. Expressions are parsed into a tree, whose root node is attached in the appropriate location along the flow of the program or control structure, as appropriate.

2.2.3 Code generation

Java code After the parser constructs an AST, the root of the tree is passed to code generation procedures to generate Java source code or Jasmin opcode. Since the FORTRAN source is assumed syntactically correct, the tree is recursively traversed without formal type-checking. The traversal is done in a single pass to generate Java source.

Jasmin opcode For Jasmin source, two passes are done: the first to assign opcode by type and context, the second pass to emit opcode. Jasmin opcode differs from from Java source primarily in 3 different ways: (1) The operator syntax is postfix instead of infix; (2) branching must be handled explicitly; and (3) arithmetic operations are performed by type specific instructions, that is different instructions are used to add integers than to add floats.

The most challenging aspect of generating opcode for jasmin is correctly implementing execution branching. Branching takes the form `jump -> label` where the jump may be a result of a comparison of two values on the stack, or simply a `goto` statement. Labels are the target of all jumps. Different control flow structures have different requirements for labeling. Appendix C illustrates branching constructions in Jasmin generated by the f2jas program.

3 Using the f2j compiler

The f2j system currently consists of C source files that are conditionally compiled into two separate executables: f2java for translating FORTRAN source into Java source, and f2jas, for translating FORTRAN source into Jasmin opcode. Compilation is controlled by preprocessor directives defined in the f2j.h header file. The defined options control whether the code is compiled into f2java or f2jas, and whether the executable translates to the netlib.blas or the netlib.lapack directory. For example, the following settings in the f2j.h header file would compile an executable, built as `f2java` in the Makefile, to compile FORTRAN to Java source code, using the BLAS headers:

```
#define JAVA 1
#define JAS  0
#define BLAS 1
#define LAPACK 0
```

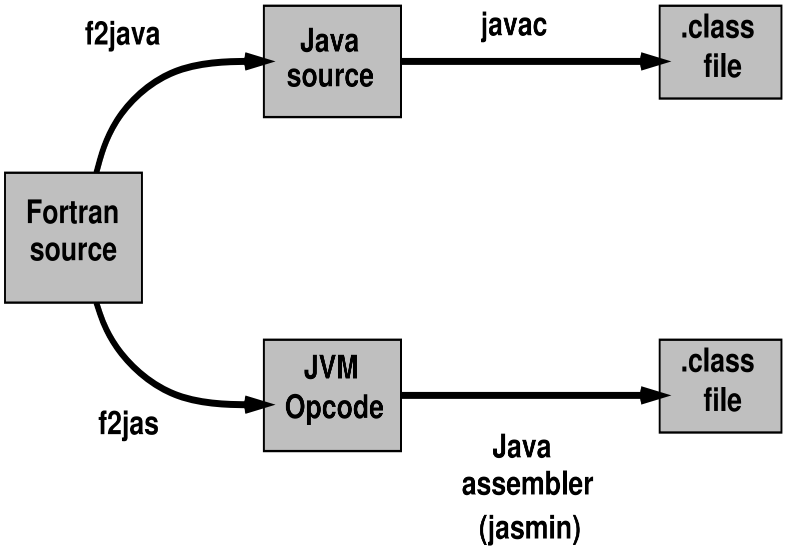

Figure 1: Translation strategies in the f2j project

Using f2java or f2jas from the command line requires only the filename of the FORTRAN program to translate. The name of the FORTRAN file is transformed into the class name, with appropriate capitalizations following established Java programming conventions. Thus, the LAPACK driver dgesv.f is translated to Dgesv.java.

4 User interface to JLAPACK

The user interface to JLAPACK will be as close to the LAPACK user interface as possible. The most significant differences will be in translating user supplied 2D arrays into 1D arrays, and typing certain variables as objects to emulate pass-by-reference. Invoking a a JLAPACK driver will then consist of declaring the appropriate types, invoking a method to transform array data, then calling the driver routine as a static method. Since the static methods in a sense shadow the class name, the syntax for calling a driver is easy to remember. For example, calling dgesv will be done in JLAPACK by Dgesv.dgesv(... *arglist* ...).

5 Current status of project

The JLAPACK project has completed Phase 1 (FORTRAN front-end) with respect to the initial design criteria. Currently, the project is partially through Phase 2. The LAPACK drivers are dependent on subroutines in the files dlamch.f and ilaenv.f, which will not compile under Phase 1 design. The FORTRAN front-end will have to be slightly extended.

The current implementation of f2j performs elementary Java source code generation, and partial Jasmin opcode generation. The generated Java code will compile if it doesn't contain goto statements, and called subroutines compile. Level 2 and 3 BLAS meet both criteria and will compile and run in the JVM.

Implicit typing of variables can be done in FORTRAN. By default, variable names starting with I, J, K, L, M or N are integer variables [12] unless explicitly typed otherwise. f2j assumes all variables in the FORTRAN source are explicitly typed.

The parser only handles one FORTRAN program unit (function, subroutine, program) per input file. This was a design decision based on the structure of most of the BLAS and LAPACK files, which in fact contain only one procedure. The file `dlamch.f`, however, contains several other subroutines and won't parse. Extending the parser to handle an arbitrary number of program units would be most convenient, but as mentioned above, memory useage could be an issue.

Currently, f2j does not do any memory recovery. All allocated memory is kept by the program and returned to the operating system only when the program exits. For the BLAS and LAPACK translation, this is not a large issue. Any future extensions to f2j, such as files containing an arbitrary number of routines, should address this issue.

An early design decision resulted in DATA and SAVE statements not being implemented in the parser or code generator. Not having SAVE statements means that the critical routines in dlamch.f will not compile, even when all the subroutines in the file are in separate source files. Since these routines are called as static methods from a number of LAPACK drivers, the javac compiler must be able to build the required class file or have the class file in the path to compile a class file for that driver. DATA statements are perhaps not as critical to implement as the SAVE statement. However, none of the existing test routines for the FORTRAN BLAS and LAPACK will compile without the DATA statement. Extending f2j to handle DATA would probably be easier than rewriting all of the test rouitnes.

6 Recommendations

Implement DATA, SAVE, and enough I/O to get the BLAS and LAPACK test routines, dlamch.f and ilaenv.f to translate.

Merge f2jas and f2java into a single executable using command line switches to control output.

Pass a COMMENT token through the parser to emit the original FORTRAN comments into the translated Java and Jasmin files.

The parser generator `yacc` was developed in the 1970's and reflects the programming practices of its day. For instance, the actual parser produced by `yacc` is not easily readable due to the number of goto statements. As the parser takes no arguments, it must interact with its associated lexer using global variables. The extent of global variable use in the f2j compiler has reached a point of diminishing returns. The code generation routines should be rewritten to reference a structure that keeps count of all relevant variables such as the name of the current program, line numbers, label numbers, etc.

Rewriting the type assignment and code generation routines for emitting Jasmin opcode involves determining exactly how many global variables are used, writing an appropriate structure and passing the structure along through recursive calls. The structure should be declared and initialized in the initialization routine called by main() before the parser starts to work.

Most of the work to date on the f2j system has been designed from the "bottom up." Functions performed by the lexer, parser and code generator were factored into independent procedures and implemented as building blocks to constuct the compiler. At the present point, midway through Phase 2, user interface design issues necessitate a "top down" approach, that is, designing JLAPACK from the potential users point of view.

One of Java's strengths is that its numerical syntax is similar to C, FORTRAN, and BASIC. Since the mechanics of implementing array access differ between Java and FORTRAN, it seems at once natural and desirable to sheild JLAPACK users from the internal workings of the JLAPACK code. Two ways of doing this are: (1) provide completely new wrappers to the driver routines using object-oriented conventions; (2) provide the user with a data specification and a set of auxilary routines. The specification would expose the user to the structure expected by JLAPACK for data. The auxiliary routines would allow the user to ignore the specification and transform Java 2D arrays in row order to 1D column order arrays. Option (1) is preferable from a design standpoint, but option (2) is much easier to implement, parallels existing LAPACK documentation, and should expose pitfalls to be avoided in a later implementation of (1).

7 Summary

FORTRAN still excels for numerical programming, and is not likely to be challenged anytime in the foreseeable future. Indeed, more powerful versions of the FORTRAN (HPF, F90) language have been developed. The numerical libraries originally developed in FORTRAN, such as BLAS and LAPACK, are *de facto* reference implementations of specific numerical algorithms.

Translating the FORTRAN directly to Java probably won't provide optimal execution speeds. However, it is a convenient first step. The issue addressed with JLAPACK is not whether it is possible to derive algorithms implemented in Java that provide the same efficiency as existing algorithms written in FORTRAN. This hasn't been resolved. In some cases, a different algorithm, derived to take advantage of Java's strengths, may provide near FORTRAN speeds when used with a JIT or compiled to native code. The issue is "how do we express, in Java, algorithms that are well known and understood, reliable, efficient and thoroughly debugged, currently written in FORTRAN." The f2j compiler is a first step in this direction.

Since these algorithms are applicable for a broad range of problems over a broad range of scales, providing reliable implementations in other languages such as C and Java provides a great benefit to the numerical computing user community. While numerical analysts may find FORTRAN the most efficient language for algorithm development, engineers and scientists in other disciplines may need to use a different language, such as Java, for application development. The numerical algorithm, instead of being the point of the program, is a tool useful for accomplishing its specified task.

The f2j compiler provides an excellent base upon which to build a more general compiler that translates a larger subset of FORTRAN into Java. For performance reasons, it may still be necessary to have some user control over how variables are passed and arrays accessed, but there are no formal obstacles, other than implementing the EQUIVALENCE statement. Such a tool could also perform code restructuring using the information implicit in the abstract syntax tree constructed during parsing. The popularity of the *f2c* translator indicates that *f2j* will be a popular and useful tool.

Acknowledgments

Clint Whaley was very helpful working out out array access issues. Susan Blackford helped with key concepts in the LAPACK software library such as variable referencing and the role of the machine constants (dlamch) subroutines. James Giles of Ricercar Software provided critical parts of the algorithm (Sale's) used to properly lex FORTRAN. John Levine provided an implementation of a FORTRAN lexer that was useful for designing the current lexer. Josef Grosch (CoCoLabs) provided a long list of ambiguities in the FORTRAN 90 specification, which helped construct the f2j grammar and would be very useful for any extension of the f2j compiler.

References

[1] I. Graham. *Object Oriented Methods.* Addison-Wesley, Berkeley, CA, 2d edition, 1994.

[2] L. E. Malvern. *Introduction to the Mechanics of a Continuous Medium.* Prentice-Hall, Englewood Cliffs, New Jersey, 1969.

[3] J. C. S. Long, J. S. Remer, C. R. Wilson, and P. A. Witherspoon. Porous Media Equivalents for Networks of Discontinuous fractures. *WRR*, 18:645–658, 1982.

[4] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5:308–325, 1979.

[5] J. Dongarra, J. Du Croz, S Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–32, 1988.

[6] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[7] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Second Edition.* SIAM, Philadelphia, PA, 1995.

[8] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. A Fortran-to-C converter. Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[9] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, 1988.

[10] S. I. Feldman. Implementation of a portable Fortran 77 compiler using modern tools. *ACM SIGPLAN Notices*, 14:8:98–106, 1979.

[11] American National standards Institute. American National Standards Institute programming language FORTRAN. X3.9-1978, ANSI, New York, New York, 1978.

[12] M. Boillot. *Understanding FORTRAN 77 with Structured Problem Solving.* West Publishing Company, New York, 2d edition, 1987.

A Sale's algorithm

Sale's algorithm (Giles, pers. communication) was published in CACM in the sixties - with an update for Fortran 77 republished in the eighties. Sale's algorithm requires a prepass of each statement to determine whether or not the statement begins with a keyword. The prepass is simple and can be done while scanning to remove comments and white space, and catenating continuation statements together, prior to normal lexical processing.

During the prepass, any characters in a Hollerith, character string (that is, between quotes or apostrophes), and any characters between matching parenthesis are ignored. Those characters are irrelevant to the purpose of the prepass. Of the remaining characters, the scanner must keep track of whether there are any equal signs (=) and whether there are any commas (,). Given the results of the prepass, the lexer should work according to the following rules, illustrated with examples.

1. If there was neither a comma nor an equal sign, the statement must begin with a keyword:

   REAL X No comma or equal sign, so a keyword must be first and lexer should find the REAL keyword;

   FORMAT(I5,F10.3) No comma or equal sign (none outside parenthesis), lexer should find keyword FORMAT.

2. If there was an equal sign, but not a comma, the statements must *not* begin with a keyword:

   REAL X = 5 No comma, but an equal sign, so no keyword is allowed, lexer should find identifier REALX;

   FORMAT(I5) = 7 An equal sign, lexer finds identifier FORMAT;

   DO 10 I = 1.10 Famous, no comma but equal sign is present, identifier DO10I found.

3. If there was a comma (equal sign or not), the statement must begin with a keyword:

   DO 10 I = 1, 10 Both comma and equal sign, keyword DO found.

However, there are still exceptions, This is resolved by collecting more information during the Sale's prepass. If the statement contains parenthetical lists or expressions, look at the first non-blank character after the close of the first parenthetical list/expression and record whether it's a letter. Now add two more rules to the lexer:

4. If there was a letter following a parenthetical, then the statement must begin with the keyword IF (very specific rule).

   IF(LOGFLG) X = 5.0 Has an equal sign, but begins with keyword!

5. If there is such a letter, but no equal sign, the statement must end with the keyword THEN (another very specific rule).

   IF(LOGFLG) THEN No comma or equal, so IF keyword is found, but there is a second keyword.

Another lexically ambiguous situation occurs with the FUNCTION keyword. To resolve this, it must be known whether this is the first statement of a procedure, or whether it's a subsequent statement. If this is the first statement in the source, or if it's the first statement after an END statement, then the form with an identifier inside the parenthesis is a FUNCTION declaration. In all other circumstances, the statement declares an array.

6. This will always declare an array.

   INTEGER FUNCTION A(5) An array declaration FUNCTIONA of length 5 (this is not ambiguous).

7. If this is the first statement in the source, or if it's the first statement after an END statement, a function is declared.

   INTEGER FUNCTION A(I) Either an array or a function is declared.

Except for the keywords inside of an I/O control list (not implemented in f2j), these rules specify how to find all Fortran 77 keywords. Except for THEN and FUNCTION, all non-I/O keywords must be the first token of the statement they're in (with the caveat that they can be the first token of the sub-statement controlled by a logical IF - to which rules 1 to 3 still apply).

Sale's algorithm resolves many of the issues that are frequently cited as major difficulties of lexically scanning Fortran. The algorithm can be implemented during a prepass to quickly identify most of the information that would normally require lookahead when during lexing. From there on, FORTRAN can be quickly and efficiently parsed with the same compiler tools used for more modern languages. The algorithm is easy to code, fast, and the rest of Fortran's syntax is fairly reqular once resolved in the lexer.

B Timing array index operations

Three timing loops were written in Java to investigate the relative execution speed resulting from initializing a square matrix, $n = 500$. The first loop initialized a 2D array, the second a 1D array with row order indexing, the third a 1D array with column order indexing. In the third loop (column order), the index product term was assigned to a dummy variable between the outer and inner loops. The timing was done by storing the system time (in milliseconds) before the loop, then taking the difference with system time returned after the loop. Garbage collection was forced before each timing call in an attempt to provide an identical execution environment for each loop.

The results of two experiments of 32 trials each are shown in Table B. The first experiment initialized arrays with the integer constant 1; the second with a double constant generated by raising a double to the power of another double. The 1D column order was the fastest, followed by the 2D and the 1D row order. The time differences reflect the both the number of statements and the time required to execute statements initializing the matrix within the inner loop. Disassembling the class file for the integer testing code (ArrayOps.java) shows that the 2D and 1D column order matrix indexing requires 6 instructions, while the 1D row order requires 8, due to the product term located within the index. Differences between the 2D and 1D column order are assumed due to variations in time required to execute different JVM instructions.

	2D	1D (row)	1D (column)
integer	480	592	462
double	2127	2268	2116

Table 1: Execution speed of array index operations depends on the number and type of instructions required to index.

As would be expected, the time differences between intializing an integer and double is significant only in a relative sense. The absolute values of the differences are very similar. A better timing loop would measure time required for a matrix-matrix operation such as $C = AB$.

The following Java source code implements timing for index operations in two dimensional arrays that are accessed by different indexing methods. Although initializing array elements is a simple operation, the code could easily be extended to implement matrix-matrix operations.

```
/*  This class performs a some simple timing for array
    operations.
    */
//
```

```
import java.lang. *;
public class ArrayOps
{

    public static void main (String[]args)
    {
       System.out.println ("  2D   1D   1D2");
       for (int i = 0; i < 33; i++)
         {
              twoD ();
              oneD ();
              oneD2 ();
              System.out.println ();
         }
    }

    public static void twoD ()
    {
       System.gc ();
       long time = System.currentTimeMillis ();
       int i, j;
       double[][] A = new double[500][500];
       for (i = 0; i < 500; i++)
         {
              for (j = 0; j < 500; j++)
               {
                   A[i][j] = Math.pow (3.14159, 2.718);
               }
         }
       System.out.print ("  " + (System.currentTimeMillis () - time));
    }

    public static void oneD ()
    {
       System.gc ();
       long time = System.currentTimeMillis ();
       int i, j, LDA;
       double[] A = new double[500 * 500];
         LDA = 500;
       for (i = 0; i < 500; i++)
         {
              for (j = 0; j < 500; j++)
               {
                   A[i + j * LDA] = Math.pow (3.14159, 2.718);
               }
         }
       System.out.print ("  " + (System.currentTimeMillis () - time));
    }

    public static void oneD2 ()
    {
```

```
        System.gc ();
        long time = System.currentTimeMillis ();
        int i, j, LDA;
        double[] A = new double[500 * 500];
          LDA = 500;
        for (j = 0; j < 500; j++)
          {
              int k = j * LDA;
              for (i = 0; i < 500; i++)
               {
                   A[i + k] = Math.pow (3.14159, 2.718);
               }
          }
        System.out.print ("  " + (System.currentTimeMillis () - time));
    }
}                             // End class file.

//
```

The class file of the ArrayOps class was disassembled into Jasmin opcode to examine the instructions required by the JVM to implement each type of array access method.

```
;
; Output created by D-Java (mailto:umsilve1@cc.umanitoba.ca)
;

;Classfile version:
;    Major: 45
;    Minor: 3

.source ArrayOps.java
.class  public synchronized ArrayOps
.super  java/lang/Object



; >> METHOD 1 <<
.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 2
.line 13
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "  2D   1D   1D2"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
.line 14
    iconst_0
    istore_1
    goto Label2
.line 16
Label1:
    invokestatic ArrayOps/twoD()V
.line 17
```

```
    invokestatic ArrayOps/oneD()V
.line 18
    invokestatic ArrayOps/oneD2()V
.line 19
    getstatic java/lang/System/out Ljava/io/PrintStream;
    invokevirtual java/io/PrintStream/println()V
.line 14
    iinc 1 1
Label2:
    iload_1
    bipush 33
    if_icmplt Label1
.line 11
    return
.end method

; >> METHOD 2 <<
.method public static twoD()V
    .limit stack 6
    .limit locals 5
.line 25
    invokestatic java/lang/System/gc()V
.line 26
    invokestatic java/lang/System/currentTimeMillis()J
    lstore_0
.line 28
    sipush 500
    sipush 500
    multianewarray [[D 2
    astore 4
.line 31
    iconst_0
    istore_2
    goto Label4
.line 33
Label1:
    iconst_0
    istore_3
    goto Label3
.line 35
Label2:
    aload 4
    iload_2
    aaload
    iload_3
    ldc2_w 3.14159
    ldc2_w 2.718
    invokestatic java/lang/Math/pow(DD)D
    dastore
.line 33
    iinc 3 1
Label3:
```

```
    iload_3
    sipush 500
    if_icmplt Label2
.line 31
    iinc 2 1
Label4:
    iload_2
    sipush 500
    if_icmplt Label1
.line 38
    getstatic java/lang/System/out Ljava/io/PrintStream;
    new java/lang/StringBuffer
    dup
    ldc "  "
    invokenonvirtual java/lang/StringBuffer/<init>(Ljava/lang/String;)V
    invokestatic java/lang/System/currentTimeMillis()J
    lload_0
    lsub
    invokevirtual java/lang/StringBuffer/append(J)Ljava/lang/StringBuffer;
    invokevirtual java/lang/StringBuffer/toString()Ljava/lang/String;
    invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
.line 23
    return
.end method

; >> METHOD 3 <<
.method public static oneD()V
    .limit stack 6
    .limit locals 6
.line 43
    invokestatic java/lang/System/gc()V
.line 44
    invokestatic java/lang/System/currentTimeMillis()J
    lstore_0
.line 46
    ldc 250000
    newarray double
    astore 5
.line 47
    sipush 500
    istore 4
.line 48
    iconst_0
    istore_2
    goto Label4
.line 50
Label1:
    iconst_0
    istore_3
    goto Label3
.line 52
Label2:
```

```
    aload 5
    iload_2
    iload_3
    iload 4
    imul
    iadd
    ldc2_w 3.14159
    ldc2_w 2.718
    invokestatic java/lang/Math/pow(DD)D
    dastore
.line 50
    iinc 3 1
Label3:
    iload_3
    sipush 500
    if_icmplt Label2
.line 48
    iinc 2 1
Label4:
    iload_2
    sipush 500
    if_icmplt Label1
.line 55
    getstatic java/lang/System/out Ljava/io/PrintStream;
    new java/lang/StringBuffer
    dup
    ldc "  "
    invokenonvirtual java/lang/StringBuffer/<init>(Ljava/lang/String;)V
    invokestatic java/lang/System/currentTimeMillis()J
    lload_0
    lsub
    invokevirtual java/lang/StringBuffer/append(J)Ljava/lang/StringBuffer;
    invokevirtual java/lang/StringBuffer/toString()Ljava/lang/String;
    invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
.line 41
    return
.end method

; >> METHOD 4 <<
.method public static oneD2()V
    .limit stack 6
    .limit locals 7
.line 60
    invokestatic java/lang/System/gc()V
.line 61
    invokestatic java/lang/System/currentTimeMillis()J
    lstore_0
.line 63
    ldc 250000
    newarray double
    astore 5
.line 64
```

```
    sipush 500
    istore 4
.line 65
    iconst_0
    istore_3
    goto Label4
.line 67
Label1:
    iload_3
    iload 4
    imul
    istore 6
.line 68
    iconst_0
    istore_2
    goto Label3
.line 70
Label2:
    aload 5
    iload_2
    iload 6
    iadd
    ldc2_w 3.14159
    ldc2_w 2.718
    invokestatic java/lang/Math/pow(DD)D
    dastore
.line 68
    iinc 2 1
Label3:
    iload_2
    sipush 500
    if_icmplt Label2
.line 65
    iinc 3 1
Label4:
    iload_3
    sipush 500
    if_icmplt Label1
.line 73
    getstatic java/lang/System/out Ljava/io/PrintStream;
    new java/lang/StringBuffer
    dup
    ldc "  "
    invokenonvirtual java/lang/StringBuffer/<init>(Ljava/lang/String;)V
    invokestatic java/lang/System/currentTimeMillis()J
    lload_0
    lsub
    invokevirtual java/lang/StringBuffer/append(J)Ljava/lang/StringBuffer;
    invokevirtual java/lang/StringBuffer/toString()Ljava/lang/String;
    invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
.line 58
    return
```

```
.end method

; >> METHOD 5 <<
.method public <init>()V
    .limit stack 1
    .limit locals 1
.line 8
    aload_0
    invokenonvirtual java/lang/Object/<init>()V
    return
.end method
```

C JVM instructions for FORTRAN

The following tables illustrate FORTRAN syntax expressed in terms of JVM instructions.

Logical if FORTRAN logical if statements take the form of a test/statement on one line. If the test is true, the statement is executed, if false, execution passes to the statement on the next line. This is implemented in jasmin by reversing the relational operator (RO) to skip the conditional execution if true. For example, consider the following code.

FORTRAN source	Jasmin target

`if (x .lt. 3) y = 1`
`     return`
	`; Logical 'if' statement.`
	`   ldc 3         ; 3`
	`   iload 0       ; x`
	`   if_icmpge Label1`
	`   ldc 1         ; 1`
	`   istore 1      ; = y`
	`Label1:`
	`   return`

The FORTRAN source requires setting $y = 1$ if $x < 3$, then returning. In the JVM, if $x \geq 3$, we jump directly to the return at `Label 1`, else executions "falls through" the test and 1 is assigned to y before returning. One unique label is required for each logical if statement.

Do loops Implementing FORTRAN do loops is done by first initializing the loop index, testing the index against the stop value, jumping back to executable statements, then incrementing the loop counter and testing its value. The following demonstrates.

FORTRAN source	Jasmin target

<pre>   do 10 j = 1, 20
        y = y + 1
   10 continue</pre>
	<pre>; Initialize counter.
	   ldc 1 ; 1
	   istore 4 ; = j
	   goto Label2
	Label1:
	; Executable statements.
	   iload 2 ; y
	   ldc 1 ; 1
	   iadd ; +
	   istore 2 ; = y
	; Increment counter.
	   iinc 4 1 ; Increment counter j.
	Label2:
	; Compare, jump to Label1 to iterate.
	   ldc 20 ; 20
	   iload 4 ; j
	   if_icmplt Label1</pre>

For nested loops, the procedure is quite similar. The interior loops are just treated as executable statements by the exterior loop.

FORTRAN source	Jasmin target

<pre>   do 30 j = 1, 321
        do 20 i = 1, 54
          y = z - 1
   20 continue</pre>
	<pre>; do loop.
	; Initialize counter.
	   ldc 1 ; 1
	   istore 3 ; = i
	   goto Label5
	Label4:
	; Executable statements.
	   iload 1 ; z
	   ldc 1 ; 1
	   isub ; -
	   istore 2 ; = y
	; Increment counter.
	   iinc 3 1 ; Increment counter i.
	Label5:
	; Compare, jump to Label4 to iterate.
	   ldc 54 ; 54
	   iload 3 ; i
	   if_icmplt Label4
	; Increment counter.
	   iinc 4 1 ; Increment counter j.
	Label6:
	; Compare, jump to Label3 to iterate.
	   ldc 321 ; 321
	   iload 4 ; j
	   if_icmplt Label3</pre>

Each instance of a do loop requires two unique labels.

Block if FORTRAN and Java share the same behavior for block if constructions, i.e. if-else-if statements, only one test may be successfully evaluated. The remaining are skipped, execution continues at the first statement after the last else-if block, or default else if present.

FORTRAN source	Jasmin target

<pre>if (i .lt. 40) then
   i = j
else if (j .lt. 50) then
   p = 1234
else if (j .ge. 60) then
   p = q
else if (x .gt. z) then
   x = z
else
   p = d * q
endif
return</pre>
	<pre>; Block 'if' statement.
	   ldc 40 ; 40
	   iload 1 ; i
	   if_icmpge Label1
	   iload 2 ; j
	   istore 1 ; = i
	   goto Label5: ; Skip remainder.
	Label1:
	   ldc 50 ; 50
	   iload 2 ; j
	   if_icmpge Label2
	   ldc 1234 ; 1234
	   istore 4 ; = p
	   goto Label5: ; Skip remainder.
	Label2:
	   ldc 60 ; 60
	   iload 2 ; j
	   if_icmplt Label3
	   iload 5 ; q
	   istore 4 ; = p
	   goto Label5: ; Skip remainder.
	Label3:
	   iload 7 ; z
	   iload 6 ; x
	   if_icmple Label4
	   iload 7 ; z
	   istore 6 ; = x
	   goto Label5: ; Skip remainder.
	Label4:
	   iload 0 ; d
	   iload 5 ; q
	   imul ; *
	   istore 4 ; = p
	Label5:
	   return</pre>

It appears from this code that each if, else if and else require a unique label. That is, a block if requires as many unique labels as exist if, else if and else statements in the block.

D To do list

Phase 1 of the f2j compiler is complete. Phases 2 and 3 will require more programming. The following list details aspects that must be done, or would help by being done.

1. f2j currently only handles one subroutine or function at a time, not lists of subroutines or functions. This could probably be implemented rather easily, but each should be freed before parsing in new files.

2. Since I am now initializing all variables to zero, there is redundancy in the variable translated from the FORTRAN PARAMETER statement.

3. Need compiler to "automatically" figure how which package the routine should go into, and which files to import. This can be hacked by using the preprocessor, or by command line switches, etc. Also, may help to switch over to Solaris for development. That way some simple java tools can be used to control compiling, etc.

4. Write an error handling routine for yyparse to indicate the approximate location of parse errors in the input file. Lex & yacc book has example. Note that the hook for this is in the lexer; line are counted.

5. Fix the in-line RELOP action in a similar way to Name <=> NAME, etc. In fact, all in-line actions should be removed from the grammar file.

6. Add code to the 'prelex()' routine to check whether there is six spaces of white at the beginning of each non-continued statement.

7. Testing toolset: suite of tools to test the lexer, parser and functionality of translated source code. So far I have two csh scripts: `blascheck` and `lapackcheck` written to see how well src in the blas and lapack directories lex and parse.

8. Fix the shift/reduce error in the parser.

9. The JVM uses a stack, for which the number of local variables and the stack depth must be calculated in advance. There needs an algorithm derived for this, keeping the track of the following data:

   - Number of locals has be at least the same as the number of arguments, more if doubles are used, even more if with typed variables.
   - The size of the stack can be calculated by keeping a running total of operations that affect the stack.

E Known bugs f2java, f2jas

These are known bugs in f2j. There are undoubtedly others.

- Computed gotos are supposed to emit as "unimplemented", but instead come out as `label0`. In jasmin, computed gotos can be implemented as subroutines, which are supported by the JVM.
- FORTRAN character strings used as arguments to subroutine calls appear to work sometimes, but not other times. That is, one call using say, ..., 'DGFT', ... might translate to the appropriate Java code ..., "DGFT", ..., other times just a pair of commas is emitted: ...,,...