JLAPACK—Compiling LAPACK FORTRAN to Java

David M. Doolin* Jack Dongarra‡† Keith Seymour‡

June 11, 1998

Abstract

The JLAPACK project provides 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 Java applications or applets, 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 (source-to-source) compiler. The LAPACK API will be considerably simplified to take advantage of Java's object-oriented design. This 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 [2] has recorded excellent floating point arithmetic speeds (68.6 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 [3, 4], 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 provides Application Programming Interfaces (APIs) to numerical libraries from Java programs. The numerical libraries will be distributed 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 [5, 6, 7] and LAPACK [8] 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 [9], and has proven to be very popular and widely used. The BLAS and LAPACK class files will be provided as a service of the Netlib repository. f2j also provides a base for a more ambitious

*Department of Civil and Environmental Engineering, University of California, Berkeley
†Mathematical Science Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831
‡Department of Computer Science, University of Tennessee, Knoxville

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: Generating Java source and Jasmin opcode for use with the JVM from the AST.

3. Phase 3: Testing, documenting and distributing BLAS and LAPACK class files.

All phases are now complete with respect to the initial design criteria (for Java source). Significant progress has been made in the translation to Jasmin opcode, which is roughly 50% complete. The Phase 3 testing for BLAS and LAPACK resulted in some changes to the initial code design, requiring the Fortran front-end to be significantly extended.

2 Design of the f2j compiler

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 parser consists of a yacc specification, which is translated to C by bison, a parser generator. The rest of the code is written in C. The following notes provide a general overview of the f2j software design and implementation.

2.1 Translating LAPACK FORTRAN to Java

2.1.1 Basic 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 variables.

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 any arbitrary integer, say -43, in FORTRAN. Java is implemented as row major access, FORTRAN as column major 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 would dereference and pass 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 a 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 receive arguments thusly: **`(double [] A, int k, int LDA)`**, `k` indicating the offset. Elements in the subarray starting at the location `i, j` would be 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 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 during the code generation pass of f2j. Note that these are tailored to LAPACK: the generated code assumes one static method per class.

- Some functions and subroutines in BLAS and LAPACK correspond to methods intrinsic to Java. The LSAMEN procedure, which compares characters independent of case, is an example corresponding to the Java `regionMatches` method.

- 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 as flags to control the behavior of some 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 initialization and SAVE statements

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 FORTRAN77 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. Additionally, we can simplify the code generator by emitting all variables as `static`, similar to the way f2c handles variable declarations. This also means that the code generator can ignore SAVE statements since every variable is already declared `static`.

2.1.7 Methods versus functions and subroutines

f2j treats names that are not recognized as intrinsics or local variables as functions or subroutines 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 or otherwise recognized by f2j.

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 functions. Both are handled by methods in Java.

2.1.8 GOTO Translation

It is preferable to translate FORTRAN programs to Java source code rather than Jasmin opcode for many reasons, but there has been a major obstacle to doing this: the GOTO statement. The FORTRAN GOTO is hard to translate to Java source code because Java does not support a `goto` statement at all. The developers of the Java language deliberately omitted the `goto` statement because they felt it would simplify the language and eliminate some common misuses of the `goto` [10]. Their replacement for the `goto` includes the multi-level `break` and `continue` statements. This section describes our approach to translating FORTRAN to Java source code while still allowing the use of GOTOs.

For our purposes, a GOTO statement in FORTRAN falls into one of two categories: (1) one that can be translated into an equivalent Java construct free of GOTOs (such as `while`, `break`, `continue`, etc.) or (2) one that cannot be translated into such a construct.

First, we examine some examples from the first category. The following segment of code from dlamch.f shows a simulated while loop written using an IF statement and a GOTO.

```
10    CONTINUE
      IF( C.EQ.ONE ) THEN
         A = 2*A
         C = DLAMC3( A, ONE )
         C = DLAMC3( C, -A )
         GO TO 10
      END IF
```

To recognize this type of construct, f2j looks for two characteristics: (1) an IF statement with a labeled CONTINUE preceding it and (2) a GOTO statement at the end of the IF block whose target is the top of the IF block. Nested simulated while loops are recognized by pushing the label of the most enclosing IF block on a stack and comparing the destination of an enclosed GOTO with that label. The label is then popped off after emitting the IF block. The Java translation for the above loop would be:

```
while (c == one)  {
  a = 2*a;
  c = Dlamc3.dlamc3(a,one);
  c = Dlamc3.dlamc3(c,-a);
}              // Close if()
```

Frequently FORTRAN programs contain GOTO statements within DO loops. This includes many different situations, but we may roughly categorize them as follows:

- The GOTO branches to the CONTINUE statement of this DO loop.
- The GOTO branches to the CONTINUE statement of some other enclosing DO loop.
- The GOTO branches to the statement following this DO loop.
- The GOTO branches to the statement following some other enclosing DO loop.
- The GOTO branches somewhere else (this equates to the second category of GOTO statements as described above).

The first two cases correspond to Java's `continue` and labeled `continue` respectively. In Java, the labeled `continue` is used in cases where there are multiple nested loops and the programmer needs to distinguish which loop to continue. The following code segment from idamax.f shows a GOTO that can be translated to a Java `continue` statement.

```
   do 30 i = 2,n
      if(dabs(dx(i)).le.dmax) go to 30
      idamax = i
      dmax = dabs(dx(i))
30 continue
```

Detecting this kind of construct is similar to detecting a simulated while loop. Each time f2j starts generating a loop, the label of that loop's CONTINUE statement is pushed onto a stack. Then when a GOTO is encountered, f2j examines the stack to determine if the GOTO is branching to the CONTINUE statement of an enclosing DO loop. The difference between translating continue statements and simulated while statements is that with continue statements, we need to examine all labels on the stack, not just the top. Notice in the following Java translation that even though we could have generated an unlabeled `continue` statement, we chose to generate labeled `continue` statements in all cases to help ensure clarity.

```
forloop30:
for (i = 2; i <= n; i++) {
  if (Math.abs(dx[(i)- 1]) <= dmax)
    continue forloop30;
  idamax = i;
  dmax = Math.abs(dx[(i)- 1]);
}              //  Close for() loop.
```

The third and fourth cases from above correspond to Java's `break` and labeled `break` respectively. f2j does not currently detect or translate these cases, but they would be handled very similarly to the translation of continue statements. The main exception is that labeled `break` statements in Java may "break out" of any enclosing statement, while labeled `continue` statements are restricted to loops (`while`, `for`, or `do`). Thus, there may be a much wider range of constructs in which a GOTO can be converted to a labeled `break`. The following segment of code is a modified version of the previous segment from idamax.f in which the GOTO now branches to the statement following the DO loop.

```
    do 40 i = 2,n
      if(dabs(dx(i)).le.dmax) go to 50
      idamax = i
      dmax = dabs(dx(i))
40  continue
50  a = 1
```

Future versions of f2j will translate the "go to 50" to a `break` statement.

The final case from the above list (that is, the GOTO branches somewhere else) has been the most difficult to deal with - it does not correspond to any equivalent Java construct. Our goal is to restructure as many FORTRAN constructs containing GOTOs as possible into equivalent Java constructs. But there will always be some GOTOs that cannot be translated in this way. To handle these cases, we have developed a method to insert GOTO statements into Java bytecode (see figure 1).

First, we use f2java to translate the FORTRAN code to Java. GOTOs are automatically translated as calls to a dummy class. So, `go to 100` would be translated as `Dummy.go_to(100)`. The labels in the FORTRAN source are also translated as calls to the dummy class (for example, the label `100` becomes `Dummy.label(100)` in the Java source).

Next we compile the java file as usual (using javac).

At this point, we could run the resulting class file (bytecode), but instead of branching, the Dummy methods would be called. The program would, almost invariably, run incorrectly. The dummy calls act only as placeholders in the bytecode to signify where *real* goto instructions should be inserted. We have developed a bytecode translation tool to perform these insertions. Since JVM instructions have variable widths (and for other reasons), we must parse the class file in order to identify the dummy method calls. For this, we borrowed parsing code from `javab`, a bytecode parallelizing tool [11]. After the bytecode has been parsed, we scan it for calls to `Dummy.label()`, recording the label and address in a hash table. We then zero the entire instruction sequence for the method call so that the resulting bytecode does not attempt to call `Dummy.label()` (in the JVM, a zero byte corresponds to the `nop` instruction - i.e., it does nothing). On the second pass, we scan for

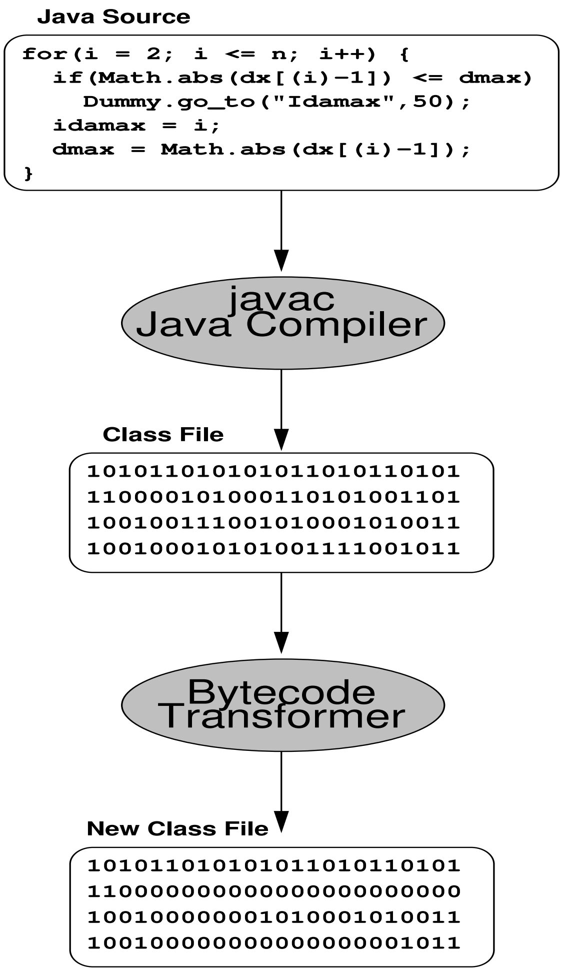

Figure 1: Translation of GOTO statements

calls to `Dummy.go_to()`. For each call, we look up the target label of the goto in the hash table to obtain the actual bytecode address of the label. Then we may replace the `Dummy.go_to()` method invocation with an actual `goto` instruction. Since the method invocation instruction sequence is longer than the goto instruction, we zero out the remaining bytes.

So far, this method has been successful in translating GOTO statements in the BLAS/LAPACK source code and testing routines. There may be some cases in which "hacking" the compiler-produced bytecode as we have done will produce unexpected results, possibly putting the JVM into an unusual state. The Java compiler generates code under certain assumptions, one example of which is that the program should not branch to a statement within a loop from outside the loop. Our GOTO translation method has the potential to violate these assumptions, although we have not yet come across a specific instance in the BLAS or LAPACK source.

2.1.9 The Need for Code Restructuring

Even though we now have a way to translate any arbitrary GOTO statement into Java source, there is still another problem caused by Java's lack of a `goto` statement. Consider the following segment of code from ddot.f:

```
      if(incx.eq.1.and.incy.eq.1)go to 20
c
      [...code for unequal increments...]
c
      ddot = dtemp
      return
c
c        code for both increments equal to 1
c
   20 m = mod(n,5)
```

The first line of code checks whether both increments are equal to 1 and if so, skips the section of code to deal with unequal increments. We can see by looking at the FORTRAN code (and so can the FORTRAN compiler) that statement 20 can be reached. However, looking at the Java source code produced by f2j, it is easy to see why the Java compiler does not recognize that fact:

```
  if (incx == 1 && incy == 1)
    Dummy.go_to("Ddot",20);

  [...code for unequal increments...]

  ddot = dtemp;
  return ddot;

  Dummy.label("Ddot",20);
  m = (n)%(5) ;
```

When the Java compiler analyzes this program, it views the call to `Dummy.go_to` as a normal method call. So, as far as javac is concerned, execution resumes with the code immediately following the method call. Since the segment of code following the method call ends with a `return` statement, javac determines that any statements following the `return` cannot be reached. Normally this would be a perfectly reasonable determination, but in this case we know that the call to `Dummy.go_to` will really cause the program to branch to statement 20.

Currently, there are a few ways to handle this. We could manually change the FORTRAN source code such that the IF expression is reversed and put the equal increments code ahead of the unequal increments code. There are code restructuring tools that can recognize and restructure these constructs automatically, removing the GOTO in the process. We could restructure the Java source code after translation (not a great idea). Future versions of f2j may have some code restructuring ability built in, so that they can generate correct Java source code without the need for manual restructuring or external tools. Our current approach is to generate one "real" `return` statement at the end of the function or subroutine, with a unique label. All other `return` statements are translated into dummy gotos, with the target being the "real" `return`. Using this approach, the java compiler views all return statements as function calls, so it no longer thinks that subsequent statements cannot be reached. This approach is much simpler and less error-prone than code restructuring.

2.1.10 DATA Statements

FORTRAN DATA statements are generated as variable declarations combined with initializations. They must be combined since all variables are now generated as static class variables and therefore cannot have separate assignment statements mixed in. For example, the following DATA statement:

```
      integer z(4)
      DATA z/1,2,3,4/
```

would be translated to Java as:

```
static int [] z = {1 , 2 , 3 , 4 };
```

Some DATA statements would be difficult to translate in this way. For example, when an array element is used in several DATA statements:

```
integer X(10)
integer Y
DATA X(3),X(8)/1,2/
DATA Y/123/
DATA X(5)/44/
```

We cannot generate declarations on a statement-by-statement basis since, in this case, we would redeclare X:

```
static int [] x = { 0, 0, 1, 0,  0, 0, 0, 2, 0, 0 };
static int y = 123;
static int [] x = { 0, 0, 0, 0, 44, 0, 0, 0, 0, 0 };
```

We could attempt to consolidate all DATA statements related to the array X and generate one declaration:

```
static int [] x = { 0, 0, 1, 0, 44, 0, 0, 2, 0, 0 };
```

Instead, we take a simpler approach. Using `static` initialization blocks, f2j can assign the values to the individual array elements one-by-one:

```
static int [] x = new int [10];
static {
  x[(3)- 1] = 1;
  x[(8)- 1] = 2;
}

static int y = 123;

static {
  x[(5)- 1] = 44;
}
```

2.1.11 EQUIVALENCE Statements

The `EQUIVALENCE` statement is one of the most difficult FORTRAN language features to translate into Java. We do not have a general solution nor do we expect to develop one, however we can support a limited form of equivalence as long as the following two conditions are met:

- The data types of the variables to be equivalenced must be exactly the same.
- If the variables are arrays, the indices, if present, must be 1.

The second condition prohibits the beginning of one array being equivalenced to the middle of another array. Arrays of different dimensions may be equivalenced, though, provided that they have the same data type.

2.1.12 Input/Output

Since our primary focus at this stage in the development of f2j is to produce a reliable Java implementation of BLAS/LAPACK, we have not emphasized the translation of Input/Output statements such as WRITE, READ, FORMAT, etc. However, it has been necessary to partially implement

WRITE/FORMAT and unformatted READ in order to compile and execute the BLAS and LAPACK test routines. File descriptors are ignored, as are field widths in FORMAT statements. Thus, the output often will not look exactly the same once the program is converted to Java, but it is close enough to verify that the numerical routines are working. Until the BLAS/LAPACK class libraries are tested and released, we do not plan to extend the implementation of I/O statements beyond what is necessary for testing the numerical code. To fully implement Fortran I/O in Java would probably require taking a similar approach to that used in f2c (possibly involving porting `libI77` to Java).

2.1.13 COMMON Blocks

As with Input/Output, our primary motivation for implementing COMMON blocks is to get the test routines translated. The BLAS and LAPACK source files do not use COMMON blocks, but they occur quite frequently in the test routines. Since our goal is to produce a reliable, thoroughly tested Java implementation of the BLAS and LAPACK libraries, it is very important to get the test routines running. Thus, f2j must provide some basic level of translation for COMMON statements.

The following segment of code from dblat1.f, the level 1 BLAS tester, illustrates a typical use of the COMMON statement:

```
      INTEGER          ICASE, INCX, INCY, MODE, N
      LOGICAL          PASS
      COMMON           /COMBLA/ICASE, N, INCX, INCY, MODE, PASS
```

Each COMMON block in the Fortran source is translated to a separate Java class:

```
public class dblat1_combla
{
  static int icase = 0;
  static int n = 0;
  static int incx = 0;
  static int incy = 0;
  static int mode = 0;
  static boolean pass = false;
}
```

Multiple COMMON blocks are supported and variables need not have the same names in each common block declaration. However, if the variables names do differ, f2j attempts to merge the names into one. Thus, if one common block has many different declarations, the merged names could become large. This is generally not a problem with the BLAS and LAPACK testers, but could become cumbersome with other code. Eventually, we may change to a different scheme, in which we choose one set of variable names to use consistently for every common block declaration.

2.1.14 Reserved words in Java

The Java language has over 50 reserved words, so it is inevitable that a Fortran program will have a variable with the same name as a Java keyword. Such a variable name cannot be retained because the Java source would not compile. Therefore, f2j maintains a table of all the Java keywords. When it comes across a variable name matching one of these keywords, it must transform the name into a new unique name that does not conflict with any Java keyword or any other currently defined variable name. Since f2j currently generates *all* Fortran variable names in lowercase, any conflicting name can be transformed by converting the first letter to uppercase (e.g. `try` becomes `Try`).

2.1.15 Object Wrappers and Optimization

As mentioned in section 2.1.1, we must encapsulate primitives in object wrappers in order to emulate pass-by-reference. However, wrapping *every* scalar would result in much lower performance than using the primitive types, but it is simple to implement in the code generator and it allows f2j to generate correct Java code automatically. Looking at the LAPACK source code, it is apparent that most of the scalar parameters are not modified within the routines. Our experience shows that a substantial performance gain can be realized by optimizing the use of object wrappers in the generated code. To perform this "optimization", we added another pass to the translator. This extra pass starts at the head of the abstract syntax tree and analyzes every variable in each program unit to determine which variables must be wrapped. The determination is made based on the following rules:

- If the variable is a parameter to the current program unit and it is on the LHS of an assignment statement, then it must be wrapped.
- If the variable is a parameter to the current program unit and it is an argument to a READ statement, then it must be wrapped.
- If the variable is an argument to some function and that function modifies the corresponding formal parameter as described in the first two rules, then it must be wrapped.

To optimize a given program unit in this way requires first optimizing all units that this one calls because we need to know whether the function parameters are wrapped or not.

Wrapping scalars in objects brings up a parameter passing issue. Consider the following function declaration:

```
double precision function ddot(n,dx,incx,dy,incy)
double precision dx(*),dy(*)
integer n,incx,incy
```

Assuming for a moment that `n`, `incx`, and `incy` are all modified within ddot, the declaration would be translated into Java as:

```
public static double ddot (intW n, double [] dx, int _dx_offset, intW incx,
                                   double [] dy, int _dy_offset, intW incy)
```

As the Java code illustrates, the integer arguments are translated to object references to provide the pass-by-reference functionality discussed previously (`intW` is the name of the integer wrapper). The two double precision vectors are translated as <*array reference, offset*> pairs (see section 2.1.2). Let us suppose that in some call to `ddot`, some of the integer parameters are constants. For example:

```
X = DDOT(5,SX,1,SY,1)
```

To correctly translate this function call, f2j must know the data types of the arguments that `ddot` expects so that the constants can be wrapped in the appropriate objects. This would not be such a problem in languages like FORTRAN and C, but with Java's strict typechecking, the resulting Java code will not compile unless all the data types match exactly. This, and other issues, necessitated the integration of a simple type analysis phase into f2j.

The previous paragraph illustrated the need for checking function arguments when passing constants, but that is not the only case in which typechecking is required. Let us consider the example FORTRAN subroutine call from section 2.1.2, `matrixop(A(i,j,), LDA)`. But in this case, imagine that `matrixop` is actually defined as follows:

```
SUBROUTINE MATRIXOP(A, LDA)
DOUBLE PRECISION A
INTEGER LDA
```

Given this definition, the translation of the subroutine call provided in section 2.1.2 (`matrixop(A, i+j*LDA, LDA)`) would be incorrect and the resulting Java code would fail compilation. If f2j did not know the data types of the arguments expected by `matrixop`, it would have to assume that an array subsection was expected and pass the <*array reference, offset*> pair, as shown above. However, if f2j can determine that `matrixop` expects scalar arguments, the subroutine call can be generated correctly by passing the array item itself: `matrixop(a[i+j*LDA],LDA)`.

Java only passes objects and arrays (which technically are considered objects) by reference. Thus, f2j could implement scalar pass by reference in either of two ways: (1) by wrapping the scalar element in an object or (2) by using a single element array containing the scalar value. Some quick timings indicate that accessing a class element is around 10% faster than accessing an array element, while accessing a primitive is around 20% faster than accessing a class element. Examining the instructions produced by the Java compiler (Sun JDK 1.1) shows why this is the case. The following is the code to access the first element of an array:

```
aload_1
iconst_0
iaload
```

The code to access a class element:

```
aload_1
getfield intW/val I
```

The code to access a primitive:

```
iload_1
```

2.1.16 Passing array elements by reference

In the previous section, we mentioned an example of a Fortran call, `matrixop(A(i,j,), LDA)`, to a subroutine declared as follows:

```
SUBROUTINE MATRIXOP(A, LDA)
DOUBLE PRECISION A
INTEGER LDA
```

That example illustrated the need to determine whether an array should be passed as an <*array reference, offset*> pair or as an array element. As mentioned before, the array element should be passed. However, the Java method call we suggested, `matrixop(a[i+j*LDA],LDA)`, is not the complete answer. It does not take into account the fact that in Fortran the array element is passed by reference while in Java it is passed by value. We take a simplistic approach to solving this problem. Before the call, we encapsulate the array element in an object. That object is then passed to the subroutine. After the call, we assign the object's value back to the array element.

```
tmp = new doubleW(a[i+j*LDA]);
Matrixop.matrixop(tmp,LDA);
a[i+j*LDA] = tmp.val;
```

This would work fine for subroutines, since in Fortran there may be only one subroutine call per line. Function calls, however, may be embedded in other calls or in other constructs making such assignments inconvenient. Therefore, we create an *adapter* to the function. An adapter merely encapsulates the real function call and the assignment statements into one method. For example, instead of calling `matrixop` directly, now we call an adapter for `matrixop`:

```
matrixop_adapter(a,i+j*LDA,LDA);
```

Then we declare the adapter as follows:

```
private static void matrixop_adapter(double [] arg0 , int arg0_offset ,intW arg1 )
{
  doubleW _f2j_tmp0 = new doubleW(arg0[arg0_offset]);

  Matrixop.matrixop(_f2j_tmp0,arg1);

  arg0[arg0_offset] = _f2j_tmp0.val;
}
```

Alternately, We could have defined the matrixop method twice in the Matrixop class, with the second declaration acting as the adapter. However, depending on the parameters, there could be several adapters for one function or subroutine and we did not want to add to the size of the library any more than necessary. Placing the adapters in the calling class allows us to only generate those adapters that will be used, rather than generating all possible adapters that could ever be used.

2.1.17 Type Analysis

Section 2.1.15 mentioned the need for type analysis in f2j by giving two examples of FORTRAN code that would not translate correctly otherwise. However, f2j does not implement typechecking, *per se*. That is, not in the traditional sense of the word. It does not attempt to warn the user of any type mismatches or other semantic errors. What f2j really does is fully traverse the AST and assign type information to every node, which is then used during code generation. This usually requires propagating type information from child nodes back up the tree to the parent nodes. Once the type assignment pass has completed, the code generator can use this information to compare arguments/parameters and generate the correct function calls. In order for the code generator to compare the actual arguments in a function call to the parameters expected by the function, the function must be parsed and analyzed by f2j. This means that to generate correct Java code, it is sometimes necessary to append external functions and subroutines to the FORTRAN program that f2j is translating. This is because the function/subroutine declaration needs to be parsed and stored in a hash table. If f2j needs to generate a function call, but cannot find the parameter information in the hash table, it will "guess" about the expected data types. That is the reason we say that it is "sometimes" necessary to append the function – because sometimes f2j will guess correctly and other times it will not. To eliminate the need for appending all the functions into one file, we may extend the f2j parser to deal with multiple files provided on the command line.

Parameter passing is not the only situation in which we might need to use information gained during the type analysis phase. Consider the following segment of code from dlamch.f:

```
      INTEGER            LBETA, LT
      DOUBLE PRECISION   A, B, C, F, ONE, QTR, SAVEC, T1, T2
*
      [...]
*
      LBETA = C + QTR
*
      [...]
```

Java does not allow assigning a double precision value to an integer variable without an explicit cast. Consequently, when f2j generates an assignment statement, it needs to determine the data type of the right hand side expression and the data type of the left hand side variable. If the data types differ, f2j must generate an explicit cast, as follows:

```
int lbeta;
double c, qtr;
[...]
lbeta = (int) (c+qtr);
[...]
```

2.1.18 Functions as Arguments

FORTRAN allows passing the name of a function or subroutine as an argument to another function or subroutine. In Java, however, passing a method as an argument is not possible. The recommended way to achieve the same effect in Java is by using *interfaces*. The prototype of the method to be passed is placed in an interface. The data type of the parameter in the called method should be the name of the interface. The method to be passed must be in a class that implements the interface.

The problem with using this approach in f2j-generated code is one of naming. When compiling a function that takes another function as a parameter, f2j does not know the name of the original function, only the name of the local parameter. So f2j cannot decide how to call the method.

Instead of using interfaces, we chose to use the *reflection* mechanism built into version 1.1 of the Java language. Reflection allows a Java program to discover information about an object (such as public methods and public class variables) at run-time. The first method in all f2j-generated classes is the translated FORTRAN routine. After that, the only other methods that should be contained in the class are adapters (see section 2.1.16). Therefore, the generated Java code can get a reference to the correct method by simply using reflection to access the first method in the class.

With this technique, the code in the calling method is simple. The user just passes a new instance of the class containing the method:

```
Sort.sort(x, 0, new Comp() );
```

It is necessary to create a new instance of the class because reflection requires a reference to the object. All methods in f2j-generated classes are static and normally there is not already an instance of the class, so we create one on-the-fly.

The code in the called method is a little more complex since it has to perform the reflection. First, the routine gets a reference to the method:

```
public static void sort (int [] a, int _a_offset, Object cfun)  {

  java.lang.reflect.Method _cfun_meth = cfun.getClass().getDeclaredMethods()[0];
```

Note that the data type of the class is `Object` because f2j does not know what kind of object is going to be passed in. Before calling the method, the arguments must be placed into an array of `Object`. Since this may take several statements and the original call may be embedded in some construct like a conditional expression, f2j generates an adapter function. The first argument to the adapter is the reference to the method. The remaining arguments mirror the parameters of the original function.

```
if(cfun_methcall(_cfun_meth,a[(i)- 1+ _a_offset],a[(j)- 1+ _a_offset]) > 0)
```

Then the adapter makes the actual call:

```
  private static int cfun_methcall( java.lang.reflect.Method _funcptr,
   int _arg0 , int _arg1 )
     throws java.lang.reflect.InvocationTargetException,
            java.lang.IllegalAccessException
  {
    Object [] _funcargs = new Object [2];
    int _retval;

    _funcargs[0] = new Integer(_arg0);
    _funcargs[1] = new Integer(_arg1);

    _retval = ( (Integer) _funcptr.invoke(null,_funcargs)).intValue();

    return _retval;
  }
```

2.2 Implementation 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 [9]*

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 preserved in the translation as Java comments. The f2java executable may generate either Java source code or Jasmin opcode, depending on the command-line options (`-java` or `-jas`).

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 [12]*

Lexing FORTRAN is somewhat difficult because keywords (e.g., IF, DO, etc.) are not reserved. Thus keywords can also be used as 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 a 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 and provided in error messages to help identify the erroneous statement.

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 "continuation" 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 continuation character, the file pointer is reset for the next call to `prelex()`.

Once `check_continued_lines()` returns a complete statement, `collapse_white_space()` removes 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 initialization header file. The tables contain either keywords, types or symbols. The appropriate table is chosen by context determined from the prepass. Table scanning is accomplished 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()` function is called if there is no context for a key word, and if the character pointed 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 [13]*

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 [14] 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 [15]. 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. Global 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 lists are built in reverse, and must be switched for in-order 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 all FORTRAN program units contained in the input file. 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 and semantically correct, the tree is recursively traversed without formal semantic checking. The Java source code generation is done in two passes. The first pass performs basic type assignment and the second pass emits the source code.

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

Synopsis

`Usage: f2java [-java/-jas] [-p package name] [-w] [-i] [-s] <filename>`

The f2j system currently consists of C source files that are compiled into a single executable: `f2java`. This executable can generate either Java or Jasmin code depending on the command-line arguments: `-java` for Java source code or `-jas` for Jasmin opcode.

Using f2java from the command line requires only the filename of the FORTRAN program to translate. By default, the FORTRAN program is translated to Java source code, but the `-jas` switch will direct f2java to generate Jasmin opcode instead. 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.

The `-p` option allows the user to specify a package name for the generated source code. For example,

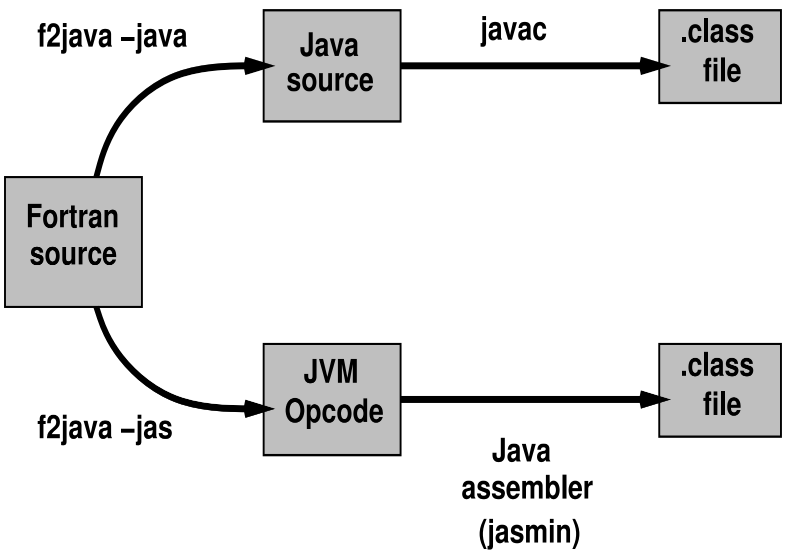

Figure 2: Translation strategies in the f2j project

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

would generate a file named File.java, as usual, with the following package specification:

```
package org.netlib.blas;
```

The package option is used when generating Java source that will be part of a library. For example, when translating ddot.f, which will be part of the Java BLAS library, we would specify "`-p org.netlib.blas`".

The `-w` option turns off the optimization of wrappers, as discussed in section 2.1.15. By default, wrapper usage is optimized. With `-w` specified, *every* scalar variable will be wrapped in an object.

The `-i` option causes f2j to generate a high-level interface to each subroutine and function. The interface provides a more Java-like API to the underlying numerical routines.

The `-s` option causes f2j to simplify the interfaces by removing the offset parameter and using a zero offset. It isn't necessary to specify the `-i` flag in addition to the `-s`.

4 User interface to JLAPACK

In JLAPACK, the API is separated into the *user* API, and the *low-level* API.

The user-level interfaces are built on top of the low-level routines. This separation provides several benefits. User supplied 2D arrays can be turned into 1D arrays, then back to 2D arrays. Offset and leading dimension arguments can be omitted to simplify the calling sequence. Certain variables must be wrapped in objects to emulate pass-by-reference, though.

The lower level interface consists of the same procedural type calls as in LAPACK. At this level, the driver routines are called as static methods. Since the static methods "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* ...).

Consult `http://www.cs.utk.edu/f2j/docs/html/packages.html` for up to date documentation on both APIs.

5 Current status of project

The JLAPACK project has now completed all three Phases (FORTRAN front-end, code generation, and testing) with respect to the initial design criteria. We are not putting much emphasis on the Jasmin code generator anymore since we can translate GOTO statements into Java source.

The current implementation of f2j performs Java source code generation, and partial Jasmin opcode generation. The generated Java code will compile as long as all dependencies compile. Levels 1, 2, and 3 BLAS, the LAPACK routines, and the BLAS/LAPACK testers all meet this criterion 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 [16] unless explicitly typed otherwise. However, f2j assumes all variables in the FORTRAN source are explicitly typed.

The parser now handles multiple program units (function, subroutine, program) per input file, however only one filename may be supplied on the command line at a time. Extending f2j to handle an arbitrary number of files would be worthwhile since the code generator needs to be able to view the function declarations of all the functions and subroutines called by a program unit.

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 the ability to specify an arbitrary number of files, should address this issue.

The current version of f2j translates certain DATA statements. Our primary motivation for implementing DATA statements was to successfully translate the BLAS testers. Consequently, the set of DATA statements f2j supports directly corresponds to the set of DATA statements used in the BLAS testers. Future versions of f2j may fully support DATA statements.

6 Recommendations

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.

Previous versions of f2j have used a single pass code generator for Java source and a two pass generator for Jasmin source. Currently, f2j uses at least two passes for both target languages, the first pass traversing the tree to determine type information and another pass generating the code. For obvious reasons, the code generation passes must remain distinct, but the type assignment passes could be combined into one common function.

Many of the FORTRAN intrinsic functions have been mapped to their Java equivalents, but there are still many remaining to be mapped.

As previously mentioned, it would be a good idea to extend f2j to allow multiple file names on the command line.

The table used to avoid conflicts between Fortran variables and Java reserved keywords should be updated to reflect the new version of the Java Development Kit (1.1).

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 construct the compiler. Midway through Phase 2, user interface design issues necessitated 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 shield 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 auxiliary 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 language (HPF, F90) 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 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 fully 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 in working 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] J. Dongarra and R. Wade. Linpack Benchmark – Java Version. [Online] Available http://www.netlib.org/benchmark/linpackjava, April 1996.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Sun Microsystems Inc. *The Java Language Environment*. Sun Microsystems, Mountain View, CA, 1995.

[11] A. Bik and D. Gannon. Exploiting Implicit Parallelism in Java. In *Concurrency, Practice and Experience*, volume 9(6), pages 579–619, 1997.

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

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

[14] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:6:607–639, 1965.

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

[16] 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 regular 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 initializing 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

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

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

Although all three Phases of the f2j compiler are complete, there are still some aspects of the system that should be done or would help by being done. The following list details some of these.

1. f2j currently only handles one file at a time. Allowing multiple files could probably be implemented rather easily, but each should be freed before parsing in new files.

2. The front-end parser and lexer are adequate for the BLAS and LAPACK libraries, but they really should be updated in order for f2j to be useful for a wide variety of libraries.
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 IF-ELSE 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 to be 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 the known bugs in f2j. There are undoubtedly others.

- 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: ...,,...
- If the last line of the input file begins with a TAB, the parser generates an error.
- The development of the Jasmin code generator has lagged far behind that of the Java code generator. There have been so many changes to the AST that the Jasmin code generator is now broken.
- Similarly, the VCG-compatible graph generator is broken, too.