VAST/f90

Fortran 90 Compiler

User's Guide
Version 3.1

Pacific-Sierra Research

Document revision record

Document Number V90B.

Edition	Date	Description
1.0	1/92	Initial release.
1.1	3/92	Minor corrections, INCLUDEs to MODULEs.
1.2	7/92	Added switches, more invocation examples.
2.0	1/93	Derived types to/from VAX-compatible structures, documentation of error detection and trace tools, Changed organization of document.
2.1	7/93	Minor improvements and corrections.
2.2	3/94	Minor improvements and corrections.
3.0	5/95	Major revision. Change document organization.
3.1	10/95	Minor reformatting.

Preface

This is a guide to the use of VAST/f90. VAST/f90 is a compiling system for Fortran 90. This manual is designed to give Fortran programmers an understanding of VAST/f90's capabilities and effective use.

Fortran 90 is the new Fortran language standard (ISO and ANSI). This User's Guide does not contain language reference information; for more information about Fortran 90, please refer to one of the many books that have been published on this subject.

An excellent source of information on Fortran 90 is the *Fortran 90 Handbook* (McGraw-Hill, 1992) by Adams, Brainerd, Martin, Smith, and Wagener. This book gives more background and examples than the standard, and still documents the complete language.

Special notices

Table of contents

1. Introduction

Compiling Fortran 90

VAST/f90 compiles Fortran 90 by translating it into Fortran 77, which is then compiled by a local Fortran 77 compiler. Using VAST/f90 is simple; you can just use the supplied `f90` driver to compile your programs. For example, if your program is `myprog.f90`, just use this command to compile it:

```
f90 myprog.f90
```

You can use whatever options you normally use for your Fortran 77 compiler and linker on the `f90` command line; they are passed to the appropriate step in the compilation process automatically.

If you just wanted to run it through VAST/f90 and see the intermediate Fortran 77 code, use:

```
vf90 myprog.f90
```

or

```
f90 -c -keep myprog.f90
```

and look at `Vmyprog.f`. This may be useful in debugging new code or in learning more about Fortran 90.

By default, VAST/f90 assumes that if the file suffix is `.f90`, the source is in free format, which is the new source form available with Fortran 90. If you have Fortran 90 programs that are in the old fixed format source form (comments starting with `C` instead of `!`, continuations via column 6 rather than with the `&`, etc.), use the `-ya` switch when you run `f90` or `vf90`, or use a `.f` file suffix. Example:

```
f90 -Wv,-ya myprog.f90
```

VAST/f90's efficient translation into Fortran 77 insures that your Fortran 90 code will run quickly.

For more information about Fortran 90 to Fortran 77 translation, see Section 3. When translating from Fortran 90 to Fortran 77, VAST/f90 converts Fortran 90 array syntax into Fortran 77 DO loops; this process is discussed in Section 4.

Further Information

Section 2 describes in more detail how to invoke VAST/f90 with the `f90` and `vf90` commands, and you probably will want to look at this section, if nothing else.

If you are interested in the specifics of Fortran 90 compilation, read sections 2 (invocation), 3 (f90 to f77), 4 (array syntax to loops) and 5 (Fortran 90 examples).

2. Invoking VAST/f90

f90 command line

To use the supplied `f90` driver for compiling Fortran 90 programs, just invoke it in the same manner as your existing Fortran 77 compiler. Put the options you normally use for your Fortran 77 compiler on the f90 command line (optimization level, etc.). If you need to pass options to VAST/f90 itself, you must prefix them with `-Wv` to distinguish them from normal compiler options. The `f90` driver automatically links in the VAST/f90 library (`libvast90.a`), which contains various functions for handling Fortran 90. The `f90` driver searches throught the user's path in order to locate `libvast90.a`.

```
f90 [f77_compiler_options] [-Wv,VAST/f90_options]
      input1 [...inputn]
```

Example 1:

To compile file `crunch.f90`, requesting optimization of the translated Fortran 77 code (`-O`), and compilation only (`-c`):

```
f90 -O -c crunch.f90
```

The object file created will be in file `crunch.o`.

Example 2:

To run `*.m` (all modules) and `sub.f90` through VAST/f90, with the input in fixed format mode (`-ya`), and compile the source output:

```
f90 -Wv,-ya *.m sub.f90
```

The executable output will be in `a.out`, as usual.

vf90 command line

VAST/f90 is executed by the command:

```
vf90 [-o output] [-l listing] [options]
  input1 [...inputn]
```

output = compilable output file. The default output file name is the input file name prefixed by `V`. The extension of the default output file will be `.f`.

options = VAST/f90 options and parameters. See Section 3.

input1... = Fortran source input files.

`vf90` invoked with no arguments prints a short usage summary, including the VAST/f90 version number.

vf90 examples

Below are some examples of using the *`vf90`* command to translate Fortran 90 to Fortran 77.

Example 1:

To run the Fortran 90 source file `crunch.f90` through VAST/f90 and get a Fortran 77 source to compile:

```
vf90 crunch.f90
f77 Vcrunch.f libvast90.a
```

The Fortran 77 output is sent by default to `Vcrunch.f`. No listing file is created. When linking the final output, you may need to link in the `libvast90.a` library.

Alternatively, you can just use the provided `f90` driver (see prior section) in this manner (it automatically links in `libvast90.a` by looking in your search path):

```
f90 crunch.f90
```

Example 2: (with modules)

To run `global.m` and `progx.f90` through VAST/f90, use:

```
vf90 global.m progx.f90
```

Module file names should be listed before the program units that reference them, because they must be processed first. Source output will be in file `Vprogx.f`.

Example 3: (fixed format)

To run `fixform.f90` through VAST/f90, where the input file uses fixed format rather than the default free format:

```
vf90 -ya fixform.f90
```

File extensions

The table below summarizes the file types that VAST/f90 may use or create.

Extension	Description
`.f`	Fixed form Fortran source.
`.inc`	Fortran INCLUDE (Fortran 90 and extended Fortran 77) (usually used to hold common blocks).
`.f90`	Fortran 90 source (free form).
`.m`	Fortran 90 module: used in place of COMMON to hold global data, may also have internal routines that operate on the data.
`.inf`	File holding Fortran 90 interface block(s): number, intent, type and attributes of the arguments to a routine -- used to check connections between program units.
`.vo`	Fortran 90 module, compiled to VAST/f90 binary form; overwritten each time the module is compiled.
`V`*`module_name`*`.inc`	Modules are converted to common blocks. Each module results in an INCLUDE file being generated. This file is then used in the Fortran 77 translation. These files can be deleted after all routines which used the specific module have been compiled into object files.

3. F90 to F77

This section discusses aspects of compiling Fortran 90 using VAST/f90. This process is automatic, and does not require intervention on your part

Translation of Fortran 90 array syntax to Fortran 77 loops is described separately in the section following this one.

The target of VAST/f90 is not limited to the Fortran 77 standard; instead, VAST/f90 targets extended Fortran 77 in most implementations. For instance, for clarity VAST/f90 on most platforms will generate loops that have no labels (DO/ENDDO), which is not standard Fortran 77 but is a feature of almost all modern Fortran 77 compilers. VAST/f90 will also generate Fortran 77 with INCLUDE statements, which are also available on most systems.

Optionally, VAST/f90 also can make use of the POINTER Fortran 77 extension (which is different from Fortran 90 Pointers, see discussion below).

VAST/f90 inspects the input source and flags Fortran 90 syntax errors.

Options

The table below shows the switches that affect the transformation of the input program. `-x` enables these switches (on), `-y` disables these switches (off).

Table 3.1 -- Transformation control switches (For Fortran 90 compilation)

Switch	Description	Default
a	Free format input source code.	on
b	Free format output source code	off
c	CM Fortran input.	off
q	Translate derived types to VAX-compatible structures	off
3	Use the F77 Pointer extension in translated output	on
4	Assume potential overlap between pointers	on
5	Generate cpp file/line number directives in output.	*

As an example, `-xb` requests that the Fortran 77 output source file be free format.

Notes on transformation switches:

a. Input file is free format. Usually, Fortran 90 input files will be free format, and this is the default. If you have a fixed format Fortran 90 file that you want converted to Fortran 77, use the `-ya` switch to request fixed format input.

b. Output file is free format. The default is fixed format output for Fortran 77 files. If you want free format Fortran 77 output, specify `-xb`.

c. Input source is Connection Machine (CM) Fortran. VAST/f90 then assumes that the input source is 132 column fixed form, and the order of arguments on array intrinsics fit CM Fortran definitions. Note: the CM Fortran intrinsics `FIRSTLOC`, `LASTLOC`, `DIAGONAL`, `PROJECT`, and `RANK` are not supported in the current version.

3. By default, on most systems VAST/f90 uses the `POINTER` extension of Fortran 77 (provided by many vendors) in the F77 intermediate code. This switch can be turned off (`-y3`) for systems that don't have this extension. Also, for systems that do have the extension, performance of code involving `POINTER`s and `ALLOCATABLE`s often improves if this feature is turned off.

4. It is possible to mask a potential recursion between a pointer and another object by pointing at identical or ovelapping areas in memory. This kind of recursion is rare in practice but it is unsafe for the compiler to assume it never happens. If it is known that it doesn't happen in the input source, use `-y4` on the command line. This will decrease execution time and memory requirements for codes making use of pointers, particularly arrays.

5. Many unix Fortran 77 compilers accept c preprocessor directives indicating source file and line number. VAST/f90 can generate these in the output as a way of relating generated source lines to original source lines. This allows runtime error messages and debuggers to refer to original source line numbers. Default is system-dependent.

The parameter `-Zrecur=nnn` can be used to specify a recursion stack size different than the default of 100.

System-dependent optimization

On some workstation platforms, VAST/f90 applies superscalar optimization to the generated Fortran 77 code, for improved runtime performance. This feature is invoked by the `-O` switch.

Typical optimizations include:

- Loop unrolling
- Loop fusion
- Loop collapse
- Cache blocking
- IF removal
- Expression optimizations

Debugging

VAST/f90 can optionally generate cpp file/line number directives for those Fortran 77 compilers that accept them. (Option `-x5`.) This allows runtime error messages and interactive debuggers to refer to original source line numbers. However, as with any optimizing compiler, there may no longer be a one-to-one correspondence between input and output source statements, and variables may have been eliminated or transformed.

Overview of Fortran 90

Coding in Fortran 90 can produce much better structured and more readable code than was possible in Fortran 77. Features of Fortran 90 include:

- *Array syntax*. You can use *triplet* notation that allows the start, end and increment of an array section to be specified. In addition, there are many new array intrinsics available. `where` statements allow conditional operations on arrays. Array syntax may be more readable than the equivalent loops.
- *Attribute form of declarations*. You can declare a list of variables to have the same attributes. Data types can be declared with the `kind` parameter.
- *New control statements*. There are `exit`, `cycle`, `case`, and `do while`. These allow you to avoid the use of confusing `goto` statements.
- *Derived types*. These allow structured data.
- *Modules*. These allow the grouping of global data (as well as procedures that operate on that data). Using modules, you can avoid the pitfalls of the old `COMMON` and `EQUIVALENCE` statements.
- *`ALLOCATE/DEALLOCATE`*. You can create and destroy data objects dynamically.
- *Interface blocks*. You can have optional arguments and keyword arguments to your procedures. Interface blocks also enable the compiler to check type and number mismatches on parameter lists and to improve the optimization of external calls. This helps to tie the program together more than was possible with older Fortrans.
- *Internal subroutines*. One level of local subroutines is allowed.
- *Pointers*. Variables can "point at" objects that are declared to be *targets*.
- *Recursive routines*. A routine may reference itself.
- *New instrinsics*. Several new intrinsics are available for string manipulation and other functions.
- *New source form*. Free format programs get away from the column-orientation of older Fortran.
- *New operators*. Symbolic relational operators are available.
- *I/O options*. You can now do non-advancing I/O, and you can specify the position of a file when you open it. These I/O features are the only part of Fortran 90 not currently available in VAST/f90.
- *Compatibility with Fortran 77*. Fortran 77 programs are valid Fortran 90 programs, so Fortran 90 maintains compatibility with older code.

Examples of various Fortran 90 constructs can be found in Section 5.

Efficiency

When coding in Fortran 90, these are some areas to be aware of, if performance of executable code is a concern:

- **Pointers**. Overuse of pointers and targets may hide true dependencies from the compiler, and result in slower code.
- **Array syntax fusion**. The compiler generates more efficient code from array syntax when there are successive conformable statements. Within a group of array syntax statements, avoid intervening scalar statements or other non-array syntax. Avoid confusing subscripting or other constructs that may inhibit the compiler's conformability analysis.
- **Passing array sections**. Passing non-contiguous array sections as procedure arguments can result in excessive data motion.
- **Allocate/Deallocate**. These operations may incur system overhead; avoid repeatedly allocating items that are used for only a short time.
- **Transformational intrinsic functions**. Certain of these, for example `SPREAD` and `RESHAPE`, may be relatively inefficient compared to DO loop equivalents.
- **Internal procedures**. Although these are intended as generalizations of the Fortran 77 statement function feature, current compilers do not always optimize references to them to the same degree as statement function references.
- **Array-valued functions**. Depending on context and implementation, array-valued function references may incur extra data motion overhead.
- **Recursion**. Generally inefficient in execution compared to non-recursive algorithms.
- **`INTENT`**. Use of interface blocks and the `INTENT` attribute for dummy arguments can help the compiler reduce unneeded operations when invoking externals.

4. Array Syntax

VAST/f90 transforms Fortran 90 array syntax into Fortran 77 DO loops; this translation of array syntax to loops is also called *scalarization*. Array syntax is one of the more important features of Fortran 90, and this section looks at it in more detail.

Features

Features of the array syntax include:

- unsubscripted arrays
- vector-valued subscripts
- triplet (colon) notation
- multi-dimensional shapes
- `WHERE/ELSEWHERE/ENDWHERE` blocks
- array constructors
- array-valued intrinsic functions
- transformational intrinsic functions
- array inquiry intrinsic functions

Array syntax expressions can appear anywhere that expressions are allowed in Fortran 77. (In some statements these would need to involve reduction operations to wind up with a scalar value).

VAST/f90 analyzes conformability of all input array expressions to insure that they meet the rules of the language.

Array expressions

Fortran 90 allows the use of array sections via the *triplet notation*, which allows a *start*, *end* and *stride* to be specified. Here are some sample expressions :

`a`	entire array
`a(n1:n2:n3)`	start:end:stride
`aa(:,:)`	entire array
`aaa(1:9:2,97:9:-1,5)`	2-dimensional section
`a * b`	element by element product
`a(1:10) + a(9:0:-1)`	element by element sum
`sin(b)`	element by element sine
`sum(a)`	sum of array elements

Conformability

Two array sections are *conformable* if they have the same shape or if one of them is a scalar. Arrays must be conformable if they are combined in an expression.

The shape of an array section is described by the *rank* and *extents*, where rank is the the number of dimensions and extent is the number of elements in a given dimension.

```
real:: a(100),aa(100,100),aaa(100,100,100)
...
a = 2.*aa(1,1:100)        conformable
a = sca                   conformable
a = aa(1:1,1:100)         non-conformable
aa = sin(aa + sca)        conformable
aa = aa + aaa             non-conformable
```

Masked operations

`WHERE` statements are used to indicate conditional array operations. The operations are done for each element corresponding to a true element in the `WHERE` clause.

```
   real, dimension(100):: a, b, c
!
   where ( a .ge. 0 ) b = sqrt(a)
!
   where ( a .eq. b )
      c = 0.
   elsewhere
      c = 1.
   endwhere
```

Intrinsic functions

There are several categories of functions that can operate on arrays in Fortran 90. Elemental functions are scalar functions which may be applied to arrays; these functions are shape preserving -- their result is the same shape as their input(s). Examples of these are `sin`, `cos`, and `sqrt`.

Transformational functions have the ability to change the shape of the result array. Examples of these are `sum`, `product`, and `reshape`.

Inquiry functions depends on a property of the array rather than the value; they return such attributes as shape or size of an array.

Here are some of the array intrinsic functions:

`SUM (ARRAY[,DIM][,MASK])`	Sum of array elements (also `PRODUCT`)
`MAXVAL (ARRAY[,DIM][,MASK])`	Maximum of array elements (also `MINVAL`)
`ALL ( MASK [,DIM] )`	Mutual 'AND' of array elements (also `ANY`)
`COUNT ( MASK [,DIM] )`	Number of true array elements.

`DOT_PRODUCT (VECTOR_A, VECTOR_B)`	Inner product of the vectors.
`MATMUL (MATRIX_A,MATRIX_B)`	Matrix multiply, or vector matrix. multiply
`TRANSPOSE (MATRIX)`	Transpose the elements of the matrix.
`PACK (ARRAY,MASK[,VECTOR])`	Where `MASK` is true, put elements of `ARRAY` into a 1-dimensional array.
`UNPACK (VECTOR,MASK,FIELD)`	Where `MASK` is true, put elements of `VECTOR` into a multi-dimensional array
`MERGE(TSOURCE,FSOURCE,MASK)`	Result is `TSOURCE` where `MASK` is true, `FSOURCE` where it is false.
`SPREAD (SOURCE,DIM,NCOPIES)`	Replicate `SOURCE` along dimension `DIM`.
`RESHAPE (SOURCE,SHAPE[,PAD][,ORDER])`	Rearrange elements of `SOURCE` into a new array.
`MAXLOC (ARRAY[,MASK])`	Find indices of max element of `ARRAY` (also `MINLOC`)
`SHAPE (SOURCE)`	Extent of `SOURCE` in each dimension.
`SIZE (ARRAY[,DIM])`	Number of elements of `ARRAY`.
`CSHIFT (ARRAY,SHIFT[,DIM])`	Circular shift of elements of `ARRAY`.
`EOSHIFT (ARRAY,SHIFT[,BOUNDARY][,DIM])`	End-off shift of elements of `ARRAY`.

Indirect addressing

Arrays can have vector-valued subscripts, as shown below; here `aa` has array section `ia(:)` as one of its subscripts.

```
      real, dimension(100,100):: aa, bb
      integer, dimension(100):: ia
   !
      bb = aa(ia(:),:)
```

Array constructors

Array constructors can be used to initialize arrays.

```
       dimension a(9)
   !
   ! array constructor example
```

```
!
    a = (/1,2,3,4,5,6,x,y,z/)
```

Translation:

```
      a(1) = 1
      a(2) = 2
      a(3) = 3
      a(4) = 4
      a(5) = 5
      a(6) = 6
      a(7) = x
      a(8) = y
      a(9) = z
```

Fusion

VAST/f90 is able to generate code for whole blocks of array assignments together (fusion). This results in much more efficient code than a line-by-line compilation, as temporaries and subexpressions can be used across statements.

Fusing statements together in this way involves examining the dependencies the array assignments represent and the conformability of successive array expressions, and determining that such fusion will not cause different answers to result.

In the example below, the loops cannot be safely fused. If the second statement referenced `a(9:19)` instead, then VAST/f90 would fuse both statements into the same loop.

```
      a(10:20) = b(10:20)
      c(10:20) = a(11:21)
```

Translation:

```
      do j1 = 1, 11
         a(j1+9) = b(j1+9)
      enddo
      do j1 = 1, 11
         c(j1+9) = a(j1+10)
      enddo
```

Some statements are split into two loops when being converted, as in this example:

```
      c(10:20) = c(9:19) + c(11:21)
```

Translation:

```
      real r1(11)
      do j1 = 1, 11
         r1(j1) = c(j1+8) + c(j1+10)
      enddo
      do j1 = 1, 11
         c(j1+9) = r1(j1)
      enddo
```

Multi-dimensional array syntax is fused where possible into nested loops. When dependencies force fusion to be abandoned along a dimension, other dimensions may still be fused.

The example below demonstrates fusion of array syntax statements into nested loops. The third statement is not conformable and is not fused with the first two.

```
      real, dimension(100,100,100):: ccc, ddd, eee
   !
   !  the next two statements are conformable
      ccc(3:m+2,3:k+2,3:l+2) =
     1  2.*ddd(4:m+3,4:k+3,4:l+3) + x
      eee(m:1:-1,k:1:-1,l:1:-1) =
     1  sqrt(eee(m:1:-1,k:1:-1,l:1:-1))
   !
   !  the next one is non-conformable with the previous
      aaa(1:10:2,:,10:1:-1) =
     1  cos ( bbb(14:5:-2,100:1:-1,1:10 )
   !
      end
```

Translation:

```
      do j3 = 1, l
         do j2 = 1, k
            do j1 = 1, m
               ccc(j1+2,j2+2,j3+2) =
     1           2.*ddd(j1+3,j2+3,j3+3) + x
               eee(m+1-j1,k+1-j2,l+1-j3) =
     1           sqrt(eee(m+1-j1,k+1-j2,l+1-j3))
            enddo
         enddo
      enddo
c
      do j3 = 1, 10
         do j2 = 1, 100
            do j1 = 1, 5
               aaa(j1*2-1,j2,11-j3) =
     1           cos(bbb(16-j1*2,101-j2,j3))
            enddo
         enddo
      enddo
```

Examples

Shown below is an example of the translation of array syntax statements involving the use of transformational functions.

```
      real, dimension(100,100,100):: aaa, ccc
      real, dimension(100,100):: bb
!
!  sum with 'mask' and 'dim' parameters
!
      bb = sum ( aaa, dim=3, mask=(ccc.le.0) )
```

Translation:

```
      do j3 = 1, 100
         do j2 = 1, 100
            bb(j2,j3) = 0
            do j1 = 1, 100
               if (ccc(j2,j3,j1) .le. 0)
     1           bb(j2,j3) = bb(j2,j3)+aaa(j2,j3,j1)
            enddo
         enddo
      enddo
```

Here is an example of `WHERE` statement translation. In the first statement, the transformational intrinsic function `SUM` must be calculated in a loop outside of the `WHERE` clause.

All of the statements excluding the first one are fused into one large loop.

```
      real, dimension(100):: a, b, c, d, e
!
!  one line WHERE with transformational
      where ( sum(a) .gt. a ) d = e
!
!  WHERE block structure
      a(1:n) = sin ( b(2:n+1) )
      where ( a(1:n) .gt. 0 )
         c(1:n) = 2.*a(1:n)
      elsewhere
         c(1:n) = 2.*e(n:1:-1)
         d(1:n) = b(1:n)*d(1:n)
      endwhere
```

Translation:

```
       r1 = 0
       do j1 = 1, 100
          r1 = r1 + a(j1)
       enddo
       do j1 = 1, 100
          if (r1 .gt. a(j1)) d(j1) = e(j1)
       enddo
c
c  WHERE block structure
       do j1 = 1, n
          a(j1) = sin(b(j1+1))
          if (a(j1) .gt. 0) then
             c(j1) = 2.*a(j1)
          else
             c(j1) = 2.*e(n+1-j1)
             d(j1) = b(j1)*d(j1)
          endif
       enddo
```

The example below shows transformational intrinsics used in scalar statements. The array calculation is done in a loop and the value used in the scalar statement.

```
!
!  Need to 'migrate' dotproduct and sum
!
      if ( dotproduct(a(1:l),b(1:l)) .gt. 0 ) then
         a = x
      elseif ( sum(a).eq.x ) then
         b = x
      endif
```

Translation:

```
      r2 = 0
      do j1 = 1, l
         r2 = r2 + a(j1)*b(j1)
      enddo
      r3 = 0
      do j1 = 1, 100
         r3 = r3 + a(j1)
      enddo
c
      if (r2 .gt. 0) then
         do j1 = 1, 100
            a(j1) = x
         enddo
      else if (r3 .eq. x) then
         do j1 = 1, 100
            b(j1) = x
         enddo
      endif
```

Finally, the example below shows the use of several keyword arguments on a transformational function.

```
      real, dimension(100,10,100):: out
      real, dimension(100,100):: a
!
! spread
!
      out = spread ( a, dim=2, ncopies=10 )
```

Translation:

```
      do j3 = 1, 100
         do j2 = 1, 10
            do j1 = 1, 100
               out(j1,j2,j3) = a(j1,j3)
            enddo
         enddo
      enddo
```

5. Fortran 90 Examples

Cycle and Exit

EXIT requests termination of the current loop, CYCLE requests the next iteration.

```
loopi: do i = 1, n
   loopj: do j = 1, m
      if ( d(i,j).eq.0 ) exit loopi
      if ( d(i,j).eq.1 ) cycle loopj
      a(i) = b(i) + d(i,j)
  end do loopj
end do loopi
```

CASEs

Cases allow a more structured test than block IF constructs.

```
IF ( ARG1 /= 0 ) THEN
   SELECT CASE (ARG1)
     CASE(1)
        ARG2 = ARG3
     CASE(2)
        ARG2 = ARG1
        ARG3 = ARG1
        SELECT CASE(ARG3+ARG2)
         CASE(3)
            WRITE(6,*)' WRONG '
         CASE(4)
            WRITE(6,*)' RIGHT '
        END SELECT
     CASE(3)
        ARG3 = 3
     CASE DEFAULT
        ARG2 = 0
   END SELECT
ENDIF
```

Derived Types

Derived types allow structuring of data. Nested derived types are allowed; derived type definitions can include references to other derived types.

If you want derived types translated into VAX-compatible structure declarations, instead of lower-level arrays, use the `-xq` switch.

```
      type stats
         integer, dimension(5) :: rbi, hrs
      end type stats
!
      type team
         character*10 player
         integer number
         type (stats) statistics
      end type team
!
      type (team) roster(9)
!
      iloop: do i = 1, 9
        write(6,50) roster(i) % player
        if ( roster(i) % player == ' ' ) cycle iloop
        do j = 1, 5
          write(6,60) roster(i) % statistics % rbi(j)
          write(6,70) roster(i) % statistics % hrs(j)
        end do
      end do iloop
!
50    format(2x, ' player: ',a10)
60    format(2x, ' rbis: ',i3)
70    format(2x, '  hrs: ',i2)
      end
```

ALLOCATE/DEALLOCATE

`ALLOCATE/DEALLOCATE` allow dynamic allocation of data items.

```
      subroutine alloc_demo
!
      real, dimension(80,90) :: a
      complex, dimension(80,90) :: b
!
      real, dimension(:,:), allocatable :: c
      complex, dimension(:,:), allocatable :: d
!
      allocate (c(20,90), d(20,90))
!
      c = cos (a(:20,:))
      d = b(:20,:)**2
      a(:20,:) = c*c - abs(d*d)
!
      deallocate (c, d)
!
      end
```

Automatic Arrays

Fortran 90 automatic arrays allow dummy argument, module, or common variables in the dimensions of local arrays in subroutines and functions. These arrays are allocated by the compiler on entry to the subprogram and deallocated on exit.

```
SUBROUTINE AUTO1 ( B, N )
REAL A(N), B(N)          (A is an automatic array.)
READ (9,*) A
B = SQRT(A) + B
WRITE (10,*) B
RETURN
END SUBROUTINE AUTO1
```

Intrinsic functions

VAST/f90 recognizes all Fortran 90 intrinsic functions and the keywords allowed for each of them. A library (`libvast90.a`) is supplied for new intrinsic functions, and should be linked into the final program (it is automatically linked in by the `f90` driver).

MODULEs

Modules allow collections of global data without the storage association of COMMON blocks; modules can also contain code related to the data.

```
MODULE NAME
   REAL A(10), B, C(20,20)
   INTEGER :: I = 0
   INTEGER, PARAMETER :: J = 10
   CHARACTER*(J) STRING
!
   CONTAINS

      SUBROUTINE ONE
         A = 1
      END SUBROUTINE ONE

END MODULE NAME
```

Variables can be renamed on the USE statement to have a different name within the referencing procedure.

```
USE NAME, ONLY: B
INTEGER A(400)
CALL ONE
```

Modules can reference other modules.

```
MODULE JQ
  REAL JX, JY, JZ
END MODULE
MODULE KQ
   USE JQ, ONLY : KX => JX, KY => JY
   ! KX and KY are local names to module KQ
   REAL KZ      ! KZ is local name to module KQ
   REAL JZ      ! JZ is local name to module KQ
END MODULE
```

POINTERs

Here is an example of Fortran 90 pointers:

```
real, pointer:: x, z
real, target:: t
!
x => t    !  pointer assignment
x = x + 1.0
z => x
z = x + 2.0
print *, z
```

Pointers can also be used with `ALLOCATE/DEALLOCATE` to allocate instances of derived types, for instance.

Fortran 90 has the `NULLIFY` statement, which allows the pointer to be nullified. The pointer can also be tested by the `ASSOCIATED` intrinsic function to see if it is currently attached.

Attribute declarations

Fortran 90 allows a new form of declarations in which multiple attributes can be specified for a given item or items in a single declaration. In addition, the `KIND` attribute can be specified, which controls the data length of the intrinsic type (real can be single or double or extended precision, for instance).

The `KIND` parameter is interpreted by VAST/f90 to be the number of bytes typically used to represent such a quantity.

```
INTEGER, INTENT(IN) :: ARG1
INTEGER, INTENT(OUT) :: ARG2
INTEGER, INTENT(INOUT) :: ARG3
LOGICAL, DIMENSION(5,5) :: MASK1,MASK2
CHARACTER (LEN=10) CHARACTER_STRING
REAL, DIMENSION(-5:+5) :: X,Y,Z
REAL (KIND(0.0D0)) A_DECLARATION
REAL, PRIVATE :: XPRIV, YPRIV, ZPRIV
```

Initial values can be specified in the declarations in a new way as well, as seen below.

```
COMPLEX :: CUBE_ROOT = (-0.5,0.866)
INTEGER, PARAMETER :: SHORT = 2
INTEGER (SHORT) K
REAL, PARAMETER :: ONE = 1.0, YY = 4.1/3.0
INTEGER, DIMENSION(3), PARAMETER :: ORDER=(/1,2,3/)
```

In addition, constants can be postfixed with a kind identifier (using an _), as in `1.0e6_8`.

User-defined operators

With Fortran 90, the programmer can define new operators which can be used in place of function calls to perform operations.

In the example below, the user-defined operator `.USER.` stands for several function calls.

```
INTERFACE OPERATOR( .USER. )
   INTEGER FUNCTION INT_USER(I,K)
   INTEGER, INTENT(IN) :: I,K
   END FUNCTION INT_USER

   REAL FUNCTION REAL_USER(R,S)
   REAL, INTENT(IN) :: R,S
   END FUNCTION REAL_USER
END INTERFACE

INTEGER I,K,L
REAL Q, R, S
LOGICAL L1, L2, L3
L = K .USER. I
Q = R .USER. S
```

Overloaded operators

With Fortran 90, you can "overload" existing operators such as `+, -, *,` etc. by defining different operations for them for different data types. You can even add meaning to the assignment operator(=).

You cannot overload an operator which is already defined intrinsically, as in *`integer`* `+` *`integer`* or *`logical`*`.and.`*`logical`* or *`real`* `=` *`integer`*.

In the example below, the `.AND.` operator is redefined for integer arguments.

```
INTERFACE OPERATOR( .AND. )
   INTEGER FUNCTION INT_AND(I,K)
   INTEGER, INTENT(IN) :: I,K
   END FUNCTION INT_AND

   REAL FUNCTION REAL_AND(R,S)
   REAL, INTENT(IN) :: R,S
   END FUNCTION REAL_AND

END INTERFACE

INTEGER I,K,L
REAL Q, R, S
LOGICAL L1, L2, L3
L = K .AND. I
Q = R .AND. S
L1 = L2 .AND. L3
```

The example below shows redefinition of the assignment operator.

```
INTERFACE ASSIGNMENT ( = )
   SUBROUTINE BIT_TO_NUMERIC(N,B)
   INTEGER, INTENT(OUT) :: N
   LOGICAL, INTENT(IN) :: B(:)
   END SUBROUTINE BIT_TO_NUMERIC

   SUBROUTINE NUMERIC_TO_BIT(B,N)
   LOGICAL, INTENT(OUT) :: B
   INTEGER, INTENT(IN) :: N
   END SUBROUTINE NUMERIC_TO_BIT

END INTERFACE

LOGICAL LL
INTEGER BB
BB = LL
LL = BB
```

Optional arguments

In order to use `OPTIONAL` arguments, you must have an interface block (either in line or from a `USE`d `MODULE` or an `INCLUDE`). Optional arguments allow calls to be made with arguments missing. Each optional argument must be declared `OPTIONAL` in the interface file and in the called routine. In the called routine, the optional argument can be tested for with the `PRESENT` intrinsic function.

```
SUBROUTINE UNO ( Z )
REAL Z
INTERFACE
   SUBROUTINE DOS ( EXTRA, TEMP, RESULT )
     REAL, INTENT(IN) :: EXTRA, TEMP
     OPTIONAL EXTRA
     REAL, INTENT(OUT) :: RESULT
   END SUBROUTINE DOS
END INTERFACE
CALL DOS ( RESULT = Z, TEMP = 272.0 )
PRINT *, Z
RETURN
END SUBROUTINE UNO
```

Internal Procedures

Internal procedures are set off by the `CONTAINS` statement. Only one level of internal procedure is allowed in the F90 Standard, so you cannot have internal procedures within internal procedures. However, module procedures may have internal procedures.

```
   SUBROUTINE ONE
   N = 2
   CALL TWO
   N = 3
   CALL THREE
!
```

```
   CONTAINS
      SUBROUTINE TWO
         PRINT *, N
      END SUBROUTINE TWO
!
      SUBROUTINE THREE
         PRINT *, N
      END SUBROUTINE THREE
!
   END SUBROUTINE ONE
```

Recursive routines

With Fortran 90, routines can call themselves if they are declared as recursive.

```
recursive integer function fact ( j ) &
   result (factx)
if ( j.eq.1 .or. j.eq.0 ) then
   factx = 1
else
   factx = j * fact(j-1)
endif
end
```

I/O Extensions

This version of VAST/f90 does not translate the few I/O enhancements available in Fortran 90. The extensions are:

- File positioning on OPEN
- Non-advancing I/O
- Additional edit descriptors

VAST/f90 issues diganostic messages whenever these unimplemented F90 features are encountered.

Other Fortran 90 features

Among other features in Fortran 90 are:

- The `RESULT` clause in Fortran 90 allows a different returnvalue variable for a function.
- Unit name on `END` statement.
- New symbolic comparison operators `<=`, etc.

Restrictions

Following is a list of restrictions and unimplemented features.

- Use of `SEQUENCE` with derived types, including:
  - Equivalence of derived type with non-conformable derived types or other variables.
  - Sequence derived types arrays in COMMON cannot be accessed in all possible ways.

```
type abc
       sequence
       integer, dimension(10) :: a,b
     end type
type(abc) da
common /.../ da(10)
...
da(j)%a(11) = ...     [one might think this refers to da(j)%b(1)]
```

  - `TRANSFER` intrinsic.

```
type uh_oh
   sequence
   real x, y
end type
type (uh_oh) bad(100)
complex c
c = TRANSFER (bad(50),c)
```

- Nonadvancing input. (Nonadvancing output is translated using the `$` edit descriptor, where available.)

- For some systems, list-directed I/O to internal file. (This is legal f90 but not f77; some f77 compilers allow, some don't.)

- For some systems, character array pointers. (Depends on f77 compiler.)

- Non-f77-pointer mode (-y3): doesn't handle character pointers or mis-aligned data.

- Recursion is limited to a stack size, with default of 100. The stack size can be increased or decreased with the `-Zrecur=`*`value`* option.

- `NAMELIST` I/O for derived types.

- `DELIM=` on `OPEN` statements.

- `RECL=` on `OPEN` statements for sequential access files.

- Use of array elements in specification (i.e. dimensioning) expressions.
- Recursive ENTRYs.
- SIZE of expression involving implied DOs with certain kinds of variable upper bounds.

```
subroutine sub(n)
isize = SIZE( (/((i,i=1,j),j=1,n)/) )
```

- Seven-dimensional array in recursive routine.
- Automatic character objects in recursive routines.
- DATA statement with vector subscript.

```
integer,parameter :: v(3)=(/3,2,1/)
integer a(3)
data a(v(:)) /1,2,3/
write(*,*) a
```

- Certain user character functions with dynamic length.
- Use of certain intrinsic functions in dimensioning expressions of dummy arrays and character variables.

```
subroutine sub(a,n,m)
dimension a(max(n,m),100)
```

- Some arguments to INQUIRE(IOLENGTH=):
  - dummy arguments
  - implied DOs
  - derived types
  - expressions involving functions
- Function argument to recursive procedure, if the function can be different at different levels of the recursion.
- DATAing of derived type variables in BLOCK DATAs; handled only if no components are arrays or structures.
- Indirect recursion.
- For some systems, possibly a few of the new edit descriptors e.g. B, depending on what extensions the f77 compiler allows.

Index