
Sanjiva Weerawarana
Ann C. Catlin
Elias N. Houstis
John R. Rice

Purdue University, enh@cs.purdue.edu
Purdue University, jrr@cs.purdue.edu

Report Number:
93-058
GENCRA\text{Y} Language Reference Manual, Version 2

Sanjiva Weerawarana
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907.

September 6, 1993

Abstract

GENCRA\text{Y} is a code translator that translates a Lisp-like language to FORTRAN 77. This manual describes the input language of GENCRA\text{Y}. The GENCRA\text{Y} input language is very similar to Common LISP, but contains extensions designed to accommodate the needs of FORTRAN.

We describe each construct by providing its input form and then explaining what FORTRAN output is produced. Several examples are provided at the end of this document to give a cohesive view of the language.
Contents

1 Introduction 3

2 Declarations 3
   2.1 Type Declarations 3
   2.2 Implicit Declarations 4
   2.3 Common Declarations 4
   2.4 External Declarations 4
   2.5 Constant (Parameter) Declarations 4
   2.6 Initial Value Declarations 5

3 Expressions 5
   3.1 Arithmetic Expressions 6
   3.2 Comparison Expressions 6
   3.3 Logical Expressions 7

4 Assignment Statements 7

5 Looping Constructs 8
   5.1 Simple Iteration 8
   5.2 Conditional Iteration 8
   5.3 General Iteration 8

6 Conditional Statements 8

7 Compound Statements 9

8 Input/Output Constructs 9

9 New Scope Constructs 9

10 Declaring Functions and Subroutines 10

11 FORTRAN Specific Constructs 10
   11.1 Calling Subroutines 10
   11.2 Labels, No-ops, and Goto Statements 10
   11.3 Pausing and Stopping Execution 11

12 When All Else Fails 11

13 Known Bugs/Problems/Limitations 11

A Example: Matrix Multiplication 13

B Example: Derivative Calculation 14

C List of Reserved Strings in GENCRAY 15
1 Introduction

GENCRAY allows one to generate FORTRAN 77 code from a LISP-like input syntax. A separate
document describes why GENCRAY is useful and how it can be used; this document only describes the
input language of GENCRAY.

The language is a LISP-style, fully-paranthesised prefix language. As much as possible, Common
LISP syntax has been used, with extensions to handle the special needs of FORTRAN. For example, we
added constructs to support the declaration of common blocks. All reserved words in the language are
matched case-insensitively.¹

In what follows, the following conventions are used:

* all reserved words are printed in typewriter font, like this
* any optional component is enclosed in ‘[‘ and ‘]’ characters.

2 Declarations

There are five types of declarations that are supported: type declarations, implicit declarations (to change
the implicit typing rules), common declarations (to declare global variables), external declarations (to
declare types of external functions), and initial value declarations (to initialize variables at compile-time).

Declarations can appear anywhere in the program and are automatically moved to the declarations
section of the closest enclosing scope in the generated program.

All declarations are based on the LISP declare construct. The basic syntax is:

\[(\text{declare } \text{decln}_1 \text{ decln}_2 \ldots \text{decln}_n)\]

where each \( \text{decln}_i \) is one of the five types of declarations. The following describes the syntax of each of
these four declarations.

2.1 Type Declarations

Type declarations are used to declare both simple and array variables. The syntax is:

\[(\text{type typename id}_1 \text{ id}_2 \ldots \text{id}_n)\]

where \( \text{typename} \) is the type that \( \text{id}_i \) are to be declared to be of. \( \text{typename} \) can be a simple type or
an array type. The simple types are given in the table below.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>FORTRAN type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>normal integer type</td>
</tr>
<tr>
<td>(integer iconst)</td>
<td>integer variables of given integer (iconst) (== 2 or 4) byte length</td>
</tr>
<tr>
<td>real</td>
<td>normal real type</td>
</tr>
<tr>
<td>(real iconst)</td>
<td>real variables of given byte length</td>
</tr>
<tr>
<td>double</td>
<td>normal double type</td>
</tr>
<tr>
<td>complex</td>
<td>normal double type</td>
</tr>
<tr>
<td>(complex iconst)</td>
<td>complex variables of given byte length</td>
</tr>
<tr>
<td>logical</td>
<td>normal double type</td>
</tr>
<tr>
<td>(logical iconst)</td>
<td>logical variables of given byte length</td>
</tr>
<tr>
<td>character</td>
<td>a single character</td>
</tr>
<tr>
<td>(character iconst)</td>
<td>character variables of given byte length</td>
</tr>
<tr>
<td>(character #)</td>
<td>character variable of unknown length</td>
</tr>
</tbody>
</table>

¹This was done because in many input generation situations, it is painful to produce the keywords in lower case.
Arrays are declared using the following syntax:

(array simple_type dims)

where simple_type is a simple type (from the previous table) and dims are any number of dimensions, where each dimension is an expression.

2.2 Implicit Declarations

GENCRAY understands and uses standard FORTRAN implicit typing rules (all variables whose name starts with either i, j, k, l, m, or n (upper or lower case) are implicitly declared as integers and all other variables are implicitly declared real). The implicit declaration allows one to change these implicit rules. Syntax:

(implicit none)
or

(implicit simple_type arg1 arg2 ...argn)

where simple_type is a simple type as before, and argi is either a single character (meaning that all variables starting with that letter are to be implicitly declared to be of the specified simple type) or (letter1 letter2) (meaning that all variables starting with any letter in the range letter1 to letter2 are to be implicitly declared to be of the specified simple type). The none form is used to instruct GENCRAY to discard implicit typing and to generate the corresponding FORTRAN statement as well.

Use of implicit none is strongly recommended as this leads to more robust programs. Implicit typing mechanisms are provided (reluctantly) simply because FORTRAN has such a capability and not because we approve of or encourage the use of such a mechanism.

2.3 Common Declarations

The GENCRAY common declaration is used to declare FORTRAN global variables (or common blocks, as they're called in FORTRAN). The syntax is:

(common [(name)] id1 id2 ...idn)

where idi are identifiers are name is the optional name of the common block. This basically places the variables idi in the common block named name (or in the blank common if name is omitted).

2.4 External Declarations

The external declaration is used to define an identifier as the name of an external procedure. The syntax is:

(external id1 id2 ...idn)

where idi are identifiers.

2.5 Constant (Parameter) Declarations

The constant declaration is used to generate FORTRAN parameter statements (i.e., statements that define constants). The syntax is:

(constant identifier expr)
where identifier is the name whose constant value is being declared and expr is the constant valued expression. The typing rules that apply to identifier are exactly the same as for any other identifier— it will be implicitly typed unless an explicit type is given. Standard aliases:

parameter for constant.

2.6 Initial Value Declarations

The data declaration is used to initialize variables to values at compile time. The syntax is:

\[
\begin{align*}
\text{(data} & \text{ } \\
& (id_1, id_2, \ldots) (arg_{11}, arg_{12}, \ldots) \\
& (id_2, id_2, \ldots) (arg_{21}, arg_{22}, \ldots) \\
& \vdots \\
& (id_n, id_n, \ldots) (arg_{n1}, arg_{n2}, \ldots)
\end{align*}
\]

where \( id_{i,j} \) are identifiers (simple or array references) and \( arg_{i,j} \) are all constant valued expressions of the form \( expr \) or \( (\text{iconst} \# expr) \), where \( \text{iconst} \) is an integer constant and \( expr \) is an expression. The first form of \( \text{arg} \) says that the value of that expression is to be assigned to the corresponding variable and the latter says that the \( expr \) is to be assigned to the corresponding \( \text{iconst} \) variables. (Thus, the latter form is a short form for the case when the same value is to be assigned to multiple variables, such as the elements of an array.) in them. Examples:

GENCRAY input:

\[
\begin{align*}
\text{(declare} & \text{ } \\
& \text{(data} \ (a \ b \ c) \ (1 \ 2 \ 3)) \\
& \text{(data} \ (a) \ (1) \ (b) \ (2) \ (c) \ (3)) \\
& \text{(data} \ ((\text{aref} \ x \ 1) \ (\text{aref} \ x \ 2) \ (\text{aref} \ x \ 3) \ (\text{aref} \ x \ 4) \ (\text{aref} \ x \ 5)) \\
& \quad \ (1.0 \ (3 \# 5) \ (10.9)))
\end{align*}
\]

GENCRAY output:

\[
\begin{align*}
\text{data} & \ a,b,c/1,2,3/ \\
\text{data} & \ a/1/, \ b/2/, \ c/3/ \\
\text{data} & \ x(1),x(2),x(3),x(4),x(5)/1.0,3\#5,10.9/
\end{align*}
\]

Note: The most general form of the data statement (the form that allows implied loops for identifiers) is currently not supported by GENCRAY.

3 Expressions

Expressions in GENCRAY are built up using constants, variables and function calls.

Numeric constants are written in either decimal form or in scientific notation. The regular expressions for the set of all valid numbers are:

\[-?[0-9]+\]
\[-?[0-9]+.[0-9]*\]
\[-?[0-9]+.[0-9]*E[0-9]*\]
\[-?[0-9]+.[0-9]*D[0-9]*\]

String constants are basically any sequence of characters enclosed in " (double quote) characters.
Variables can be either simple identifiers or array references. The syntax for array references is:

\[(\text{aref } \text{name } \text{sub}_1 \text{sub}_2 \ldots \text{sub}_n)\]

where \text{name} is the name of the array variable and \text{sub}_i are the subscripts for each of its dimensions.

Function calls are expressed using the usual LISP function call syntax:

\[(\text{name } \text{arg}_1 \text{arg}_2 \ldots \text{arg}_n)\]

where \text{name} is the name of the function and \text{arg}_i are the argument expressions.

3.1 Arithmetic Expressions

Arithmetic expressions use exactly the same syntax as in LISP. The table below summarizes arithmetic operators and the basic mathematical functions:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{e}_1 + \text{e}_2 + \ldots</td>
<td>(+ \text{e}_1 \text{e}_2 \ldots)</td>
</tr>
<tr>
<td>\text{e}_1 - \text{e}_2</td>
<td>(- \text{e}_1 \text{e}_2)</td>
</tr>
<tr>
<td>\text{e}_1 \times \text{e}_2</td>
<td>(* \text{e}_1 \text{e}_2)</td>
</tr>
<tr>
<td>\text{e}_1 / \text{e}_2</td>
<td>(/ \text{e}_1 \text{e}_2)</td>
</tr>
<tr>
<td>a^b</td>
<td>(\text{exp} a b)</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>(sqrt x)</td>
</tr>
<tr>
<td>\sin(x)</td>
<td>(\text{sin} x)</td>
</tr>
<tr>
<td>\cos(x)</td>
<td>(\text{cos} x)</td>
</tr>
<tr>
<td>tan(x)</td>
<td>(\text{tan} x)</td>
</tr>
</tbody>
</table>

where \text{e}_1, \text{e}_2, \ldots, a, b, and z are all expressions. Standard aliases:

- plus for +
- minus for -
- times for *
- quotient for /.

3.2 Comparison Expressions

These operators can be used to compare values of numerical variables or expressions.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{e}_1 \equiv \text{e}_2</td>
<td>(= \text{e}_1 \text{e}_2)</td>
</tr>
<tr>
<td>\text{e}_1 \neq \text{e}_2</td>
<td>(/= \text{e}_1 \text{e}_2)</td>
</tr>
<tr>
<td>\text{e}_1 &lt; \text{e}_2</td>
<td>(&lt; \text{e}_1 \text{e}_2)</td>
</tr>
<tr>
<td>\text{e}_1 \leq \text{e}_2</td>
<td>(\leq \text{e}_1 \text{e}_2)</td>
</tr>
<tr>
<td>\text{e}_1 &gt; \text{e}_2</td>
<td>(&gt; \text{e}_1 \text{e}_2)</td>
</tr>
<tr>
<td>\text{e}_1 \geq \text{e}_2</td>
<td>(\geq \text{e}_1 \text{e}_2)</td>
</tr>
</tbody>
</table>

where \text{e}_1 and \text{e}_2 are all expressions. Standard aliases:

- equalp for =
- neq for /=
- lessp for <
- leqp for <=
- greaterp for >
- geqp for >=.
3.3 Logical Expressions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1$ and $l_2$ and ...</td>
<td>(and $l_1$ $l_2$ ...)</td>
</tr>
<tr>
<td>$l_1$ or $l_2$ or ...</td>
<td>(or $l_1$ $l_2$ ...)</td>
</tr>
<tr>
<td>not $l$</td>
<td>(not $l$)</td>
</tr>
</tbody>
</table>

where $l_1$, $l_2$, ..., and $l$ are all logical expressions.

Logical constants true and false are specified using the following syntax:

- (true)
- (false).

4 Assignment Statements

GENCRAY supports two types of assignment statements: Simple assignments and matrix assignments.

The simple assignment statement is the usual assignment statement; that is, it is used to assign a value to a variable. A matrix assignment statement is a convenience form used to assign different values to elements of a two-dimensional array. The syntax is:

- (setq $s_1$ $s_2$ ...)

where $s_i$ is one of:

- variable expression
- variable (matrix (exp1 [exp2 ...])1 [(exp1 [exp2 ...])2 ...]).

Every row of the matrix (identified by each of the list arguments to the matrix function) must have the same number of elements in them. Standard aliases:

- setf for setq.

Examples:

GENCRAY input:

- (setq a 12.24)
- (setq c (aref foo 34 c)
- m (matrix (12 m v)
  (a (plus a (aref foo 33 c)) f)))

GENCRAY output:

```
a=12.24
c=foo(34,c)
m(1,1)=12
m(1,2)=m
m(1,3)=v
m(2,1)=a
m(2,2)=a+foo(33,c)
m(2,3)=f
```
5 Looping Constructs

GENCRAY provides a construct for generating FORTRAN do loops as well as more general constructs that generate loops using if and goto statements in FORTRAN.

5.1 Simple Iteration

The dotimes construct is similar to the LISP construct of the same name, but different. The syntax is:

\[
\text{(dotimes (var start end [step]) body)}
\]

where \( \text{var} \) is the loop variable, \( \text{start} \) is the starting value, \( \text{end} \) is the ending value (or more precisely, the loop will terminate when \( \text{var} > \text{end} \)), and \( \text{step} \) is an optional increment value. The default increment is 1. \( \text{body} \) is of course the statements that are to be executed in the loop.

5.2 Conditional Iteration

The while construct is used to generate a loop that will execute as long as some test is true. The syntax is:

\[
\text{(while expr body)}
\]

where \( \text{expr} \) is the test expression and \( \text{body} \) is the body of the loop.

This construct can translate into either a statement using the non-standard while construct of FORTRAN or to a loop using if and goto. Which one is generated is a user-settable option.

5.3 General Iteration

GENCRAY supports the LISP do statement for general iteration. The semantics supported are those of the do* construct instead of the do construct; loop variables are updated in order and not in parallel. The loop variables are "localized" by GENCRAY to ensure that LISP semantics of the do construct defining its own scope are maintained. The syntax is:

\[
\text{(do ((var1 [init1] [step1]) \ldots (varn [initn] [streetn])) (test last) body)}
\]

where \( \text{test} \) is the expression used for loop control (the loop is executed as long as the test evaluates to false), \( \text{var}_1, \ldots, \text{var}_n \) are loop variables which are initialized and stepped using \( \text{init}_i \) and \( \text{step}_i \), respectively, \( \text{body} \) is the body of the loop, and \( \text{last} \) are the statements that are evaluated when the loop test evaluates to true (i.e., when the loop is being exited).

6 Conditional Statements

GENCRAY supports the LISP if and cond statements for conditional execution. The syntax of the if statement is:

\[
\text{(if test then-clause [else-clause])}
\]

where \( \text{test} \) is a test expression that must evaluate to a logical value, \( \text{then-clause} \) is the statement executed if the test evaluates to true and \( \text{else-clause} \) is the (optional) statement executed if the test evaluates to false.

The syntax of the cond statement is:

\[
\text{(cond (test1 [body1]) (test2 [body2]) \ldots (testn [bodyn]) [(default body)])}
\]
where \(test_i\) are expressions that must evaluate to logical values, \(body_i\) are statements that are executed if \(test_i\) evaluates to true and (if present) \(body\) is executed if all the tests fail. Standard aliases:

otherwise for default.

7 Compound Statements

GENCRAY supports the \texttt{progn} construct for defining compound statements. The syntax is:

\[
\texttt{(progn body)}
\]

where \(body\) consists of any number of statements. This is useful as the then-clause of an \texttt{if} statement, for example.

8 Input/Output Constructs

GENCRAY supports only two simple I/O constructs: \texttt{read} and \texttt{write}. The syntax for \texttt{write} is:

\[
\texttt{(write \texttt{exp}_1 \texttt{exp}_2 \ldots \texttt{exp}_n)}
\]

where \(\texttt{exp}_i\) are expressions. Standard aliases:

\texttt{print} for \texttt{write}.

The syntax for \texttt{read} is:

\[
\texttt{(read \texttt{identifier})}
\]

where \(\texttt{identifier}\) is the identifier into which something is to be read. (It can be either a simple identifier or an array reference.)

The FORTRAN code generated does free format I/O.

9 New Scope Constructs

GENCRAY supports the LISP \texttt{let} construct for defining a local scope. The syntax is:

\[
\texttt{(let ([\texttt{locals}]) body)}
\]

where \(body\) consists of the executable statements of the new scope and \(\texttt{locals}\) is any number of any of the following:

\[
\texttt{(identifier \texttt{expr})}
\]

\texttt{identifier}

where \(\texttt{identifier}\) is an identifier and \(\texttt{expr}\) is an expression. The latter two forms are exactly the same; they both define an uninitialized variable. The first form defines a variable and also initializes it to the given value. Standard aliases:

\texttt{prog} for \texttt{let}.

Note: Although we have aliased \texttt{prog} for \texttt{let}, it should be noted that the semantics of \texttt{prog} and \texttt{let}
are not the same. However, for our purposes, this alias is reasonable as we do not purport to support the exact semantics of Common LISP.

10 Declaring Functions and Subroutines

FORTRAN Functions and subroutines are declared using a slight extension of the LISP defun macro. The syntax is:

\[
\text{(defun name [type] (id}_1 \text{id}_2 \ldots \text{id}_n \text{) body)}
\]

where \text{name} is the name of the function or subroutine, \text{type} is the return type (if a function is being declared), \text{id}_1, \ldots, \text{id}_n are the parameters and body is the body of the function or subroutine.

Omitting the return type \text{type} generates \text{name} as a FORTRAN subroutine. Thus, functions are generated by including a return type. When \text{type} is present, then it must be a simple type (i.e., not an array type; see the discussion on types earlier for details). Note: Requiring that a return type be specified in order to generate a FORTRAN function is a restriction from what FORTRAN requires as FORTRAN does implicit typing on the name of the function to determine its return type.

A return statement can be included in the body of the function by using the \text{return} construct:

\[
\text{(return [expr])}
\]

where the optional argument expr is used to specify alternate returns. (The alternate return feature of FORTRAN is a rarely used construct that should be avoided in almost every case.)

A special construct is available for declaring the "main" function. This is the \text{program} construct:

\[
\text{(program name body)}
\]

where \text{name} is the name of the program and \text{body} is the body of the main function.

11 FORTRAN Specific Constructs

We have defined several constructs expressly for the purpose of generating various FORTRAN constructs. This section will explain these constructs.

11.1 Calling Subroutines

Subroutines are called using a \text{call} statement:

\[
\text{(call name arg}_1 \text{arg}_2 \ldots \text{arg}_n)
\]

where \text{name} is the name of the subroutine and \text{arg}_i are the argument expressions.

11.2 Labels, No-Ops, and Goto Statements

Statement labels can be generated using the \text{label} construct:

\[
\text{(label icontst)}
\]

where \text{icontst} is an integer constant. This label will label whatever the next statement that is generated. A no-op statement can be generated using the \text{continue} construct:
A jump to a predefined label is generated using the `goto` construct:

```plaintext
(goto icanst)
```

where `icanst` is the statement label to jump to.

### 11.3 Pausing and Stopping Execution

GENCRAY provides a `pause` construct that corresponds to the FORTRAN `pause` statement. The syntax is:

```plaintext
(pause [constant])
```

where `constant` is a number or a string constant, the use of which is somewhat dependent on the FORTRAN compiler that compiles the generated code.

GENCRAY's `stop` construct is used to generate the FORTRAN `stop` statement used to stop execution of the program:

```plaintext
(stop)
```

### 12 When All Else Fails

GENCRAY provides a `literal` construct that can be used to generate FORTRAN (or other) code that cannot be generated from the GENCRAY language directly. The syntax is:

```plaintext
(literal arg1 arg2 ... argn)
```

where `argi` is any expression. However, if any `argi` is a string constant, then it is printed `without` the enclosing quotes- this allows one to generate arbitrary FORTRAN code by using string constants judiciously. In addition to expressions, two special symbols are allowed as arguments to the literal construct:

```plaintext
$tab
$n1
```

The first form is used to control indentation in the output code- using `$n1` forces the output to be tabbed up to the current indentation level. The second is used to insert a newline in the output.

### 13 Known Bugs/Problems/Limitations

- All negative numbers are enclosed in parenthesis.
- The I/O constructs are woefully inadequate.
- Language syntax allows specification (and code generation for) much more than what is valid in FORTRAN.
References


A Example: Matrix Multiplication

This example shows how one would write a matrix–vector multiplication routine using the GENCRAY language. Here is the code:

```
(defun matvec (a lda m n b res)
  (declare (type integer lda m n)
           (type (array real lda 1) a)
           (type (array real 1) b res))

  (dotimes (i 1 m)
    (setq (aref res i) 0.0)
    (dotimes (j 1 n)
      (setq (aref res i) (+ (aref res i) (* (aref a i j) (aref b j))))))
)
```

This defines a function `matvec` with six arguments. This function multiplies the \( m \times n \) matrix \( a \) and the \( n \)-vector \( b \) to produce the \( m \)-vector \( res \). The leading declaration dimension of \( a \), \( lda \), is also provided.

The arguments \( lda \), \( m \), and \( n \) are declared to be integers. The matrix \( a \) is declared to be an \( lda \times 1 \) array (according to FORTRAN's needs) and the vectors \( b \) and \( res \) are declared to be 1-dimensional arrays of length one.

The computation is specified using the `dotimes` construct. The outer loop uses the variable \( i \) as a loop counter and loops from 1 to \( m \). The inner loop uses the loop counter \( j \) and loops from 1 to \( n \). The code implements a straightforward matrix–vector multiplication algorithm.

The result of running the above input through GENCRAY is:

```
subroutine matvec (a,lda,m,n,b,res)
integer lda,m,n
real a(lda,1)
real b(1),res(1)

  do 1001 i=1,m
    res(i)=0.0
  do 1002 j=1,n
    res(i)=res(i)+a(i,j)*b(j)
  1002 continue
1001 continue
return
end
```


B Example: Derivative Calculation

Consider the following function:

\[ f(x) = \begin{cases} 
  x^2 \sin(x) & \text{if } x \geq 0 \\
  x^3 & \text{otherwise}
\end{cases} \]

Then, its derivative function can be specified as follows in GENCRAY:

```
(defun deriv real (x)
  (declare (type real x))
  (if (>= x 0)
    (setq deriv (+ (* (cos x) (expt x 2)) (* (* 2 x) (sin x))))
    (setq deriv (* 3 (expt x 2))))
)
```

Alternatively, we could have used the `cond` construct to define the input as follows:

```
(defun deriv real (x)
  (declare (type real x))
  (cond
    ((>= x 0) (setq deriv (+ (* (cos x) (expt x 2)) (* (* 2 x) (sin x)))))
    (otherwise (setq deriv (* 3 (expt x 2))))
  )
)
```

In either case, the resulting FORTRAN code is:

```
real function deriv (x)
real x

if (x .ge. 0) then
  deriv=cos(x)*x**2+(2*x)*sin(x)
else
  deriv=3*x**2
endif
return
end
```
### C List of Reserved Strings in GENCRAY

<table>
<thead>
<tr>
<th>#</th>
<th>,</th>
<th>(</th>
<th>)</th>
<th>*</th>
<th>+</th>
<th>-</th>
<th>/</th>
<th>/=</th>
<th>:</th>
<th>&lt;</th>
<th>&lt;=</th>
<th>=</th>
<th>&gt;</th>
<th>&gt;=</th>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td>aref</td>
<td>array</td>
<td>call</td>
<td>character</td>
<td>common</td>
<td>complex</td>
<td>cond</td>
<td>constant</td>
<td>continue</td>
<td>cos</td>
<td>data</td>
<td>declare</td>
<td>default</td>
<td>defun</td>
<td>do</td>
<td>dotimes</td>
</tr>
<tr>
<td>false</td>
<td>for</td>
<td>geqp</td>
<td>goto</td>
<td>greaterp</td>
<td>if</td>
<td>implicit</td>
<td>integer</td>
<td>label</td>
<td>leqp</td>
<td>lessp</td>
<td>let</td>
<td>literal</td>
<td>logical</td>
<td>matrix</td>
<td>minus</td>
</tr>
</tbody>
</table>