Languages Library

TAL Reference Summary

Abstract	This manual provides a reference summary for TAL (Transaction Application Language) for system and application programmers.
Part Number	096256
Edition	Second
Published	September 1993
Product Version	TAL C30, TAL D10, TAL D20
Release ID	D20.00
Supported Releases	This manual supports C30/D10.00 and all subsequent releases until otherwise indicated in a new edition.

Document History

Edition	Part Number	Product Version	Earliest Supported Release	Published
First	21455	TAL C20	N/A	March 1989
Second	096256	TAL C30 TAL D10 TAL D20	C30/D10.00	September 1993

New editions incorporate any updates issued since the previous edition.

A plus sign (+) after a release ID indicates that this manual describes function added to the base release, either by an interim product modification (IPM) or by a new product version on a .99 site update tape (SUT).

Export Statement Export of the information contained in this manual may require authorization from the U. S. Department of Commerce.

Examples Examples and sample programs are for illustration only and may not be suited for your particular purpose. Tandem does not warrant, guarantee, or make any representations regarding the use or the results of the use of any examples or sample programs in any documentation. You should verify the applicability of any example or sample program before placing the software into productive use.

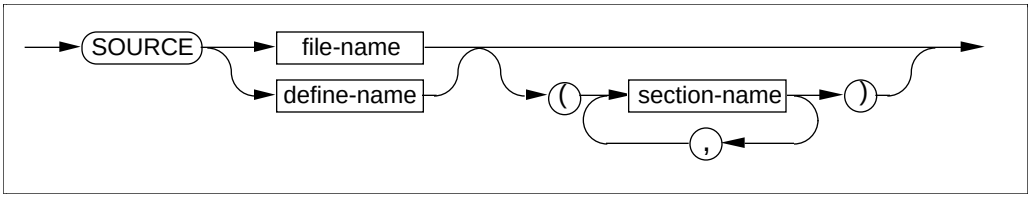
Ordering Information For manual ordering information: Domestic U.S. customers, call 1-800-243-6886; international customers, contact your local sales representative.

Contents

Reference Summary

Notation Conventions

This reference summary presents syntax in railroad diagrams. To use a railroad diagram, follow the direction of the arrows and specify syntactic items as indicated by the diagram and the term definitions that follow the diagram. Here is an example of a railroad diagram:

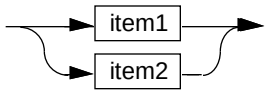


The parts of the diagram have these meanings:

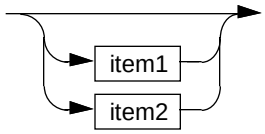
SOURCE	Specify the keyword as shown, using uppercase or lowercase.
,	Specify the symbol or punctuation as shown.
section-name	Supply the information indicated, using uppercase or lowercase.

Branching

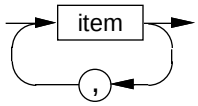
Branching lines indicate a choice, such as:



Required choice. Specify one of the items.

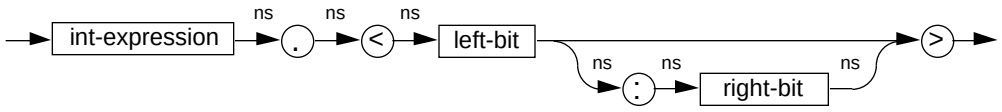


Optional choice. Specify one or none of the items.



Repeatable item. Specify one or more of the items.

Spacing Where no space is allowed, the notation **ns** appears in the railroad diagram. Here is an example of a diagram in which spaces are not allowed:



016

You can prefix identifiers of standard indirect variables with the standard indirection symbol (.) with no intervening space.

In all other cases, if no separator—such as a comma, semicolon, or parenthesis—is shown, separate syntactic items with at least one space.

Case Conventions Case conventions apply to keywords, variable items (information you supply), and identifiers. These terms appear in uppercase or lowercase as follows:

Term	Context	Case	Example
Keywords	In text and in railroad diagrams	Uppercase	RETURN
Variable items	In railroad diagrams	Lowercase	file-name
Variable items	In text	Lowercase italics	*file-name*
Identifiers	In examples	Lowercase	`INT total;`
Identifiers	In text	Uppercase	TOTAL

Running the Compiler

To run the compiler, issue a compilation command at the TACL prompt. For example, you can compile the source file MYSOURCE and have the object code sent to the object file MYOBJECT as follows:

```
TAL /IN mysource/ myobject
```

You can specify the following options in the compilation command.

IN File Option

In the compilation command, the IN file is the source file. You can specify a file name or a TACL DEFINE as described in Appendix E. In the preceding example, the IN file is MYSOURCE.

The IN file can be an edit-format disk file, a terminal, a magnetic tape unit, or a process. The compiler reads the file as 132-byte records.

If you omit the IN file and the TACL product is in interactive mode, the default file is your home terminal. In noninteractive mode, the default file is the TACL command file. For information about the TACL product, see the *TACL Reference Manual*.

OUT File Option

The OUT file receives the compiler listings. The OUT file can be a disk file (not in edit format), a terminal, a line printer, a spooler location, a magnetic tape unit, or a process.

In an unstructured disk file, each record has 132 characters; partial lines are filled with blanks through column 132. You can specify a file name or a TACL DEFINE name. The file must exist before you specify its name for the OUT file. You can create the file by using the File Utility Program (FUP). The following example specifies a file named MYLIST as the OUT file:

```
TAL /IN mysource, OUT mylist/ myobject
```

The OUT file is often a spooler location, in this case, $S.#LISTS:

```
TAL /IN mysource, OUT $s.#lists/ myobject
```

If you specify OUT with no file, you suppress the listings:

```
TAL /IN mysource, OUT/ myobject
```

If you omit OUT and the TACL product is in interactive mode, the listings go to the home terminal. In noninteractive mode, the listings go to the current TACL OUT file:

```
TAL /IN mysource/ myobject
```

TACL Run Options

You can include one or more TACL run options in the compilation command, such as:

- ☐ A process name
- ☐ A CPU number
- ☐ A priority level
- ☐ The NOWAIT option

For example, you can specify CPU 3 and NOWAIT when you run the compiler:

```
TAL /IN mysource, CPU 3, NOWAIT/ myobject
```

Another run option you can specify is the MEM (memory) option, but the compiler always uses 64 pages.

Target File Option

The target file is the disk file that is to receive the object code. You can specify a file name or a TACL DEFINE name as described in Appendix E of the *TAL Programmer's Guide*.

Previous examples sent the object code to a disk file named MYOBJECT:

```
TAL /IN mysource/ myobject
```

If you omit the target file, BINSERV creates a file named OBJECT on your current default subvolume. If an existing file has the name OBJECT or the name you specify, BINSERV purges the file before creating the new target file. If the existing file is secured so BINSERV cannot purge it, BINSERV creates a file named ZZBI*nnnn*, where *nnnn* is a different number each time.

Compiler Directives

You can include one or more compiler directives in the compilation command. Precede the directives with a semicolon and separate them with commas.

You can control the compilation listing. For example, NOMAP suppresses the symbol map, and CROSSREF produces cross-reference listings:

```
TAL /IN mysource/ myobject; NOMAP,CROSSREF
```

You can specify any directive in the compilation command except the following, which can appear only in the source file:

ASSERTION
BEGINCOMPILATION
DECS
DUMPCONS
ENDIF
IF
IFNOT
PAGE
RP
SECTION
SOURCE

The following directives can appear only in the compilation command:

EXTENDTALHEAP
SQL with the PAGES option
SYMBOLPAGES

Completion Codes Returned by the Compiler

When the compiler compiles a source file, it completes the compilation normally or stops abnormally. It then returns a process-completion code to the TACL product indicating the status of the compilation. The process-completion code values are:

Code	Termination	Meaning
0	Normal	The compiler found no errors or unsuppressed warnings in the source file. (Warnings suppressed by the NOWARN directive do not count.) The object file is complete and valid (unless a SYNTAX directive suppressed its creation).
1	Normal	The compiler found at least one unsuppressed warning. (Warnings suppressed by the NOWARN directive do not count.) The object file is complete and valid (unless a SYNTAX directive suppressed its creation).
2	Normal	The compiler found at least one compilation error and did not create an object file.
3	Abnormal	The compiler exhausted an internal resource such as symbol table space or could not access an external resource such as a file. The compiler did not create an object file.
5	Abnormal	The compiler discovered a logic error during internal consistency checking or one of the compiler's server processes terminated abnormally. The compiler did not create an object file.
8	Normal	The compiler could not use the object file name you specified, so it chose the name reported in the summary. The object file is complete and valid.

Keywords

Keywords have predefined meanings to the compiler when used as shown in the syntax diagrams in this manual.

Reserved Keywords

The following keywords are reserved by the compiler. Do not use reserved keywords for your identifiers.

AND	DO	FORWARD	MAIN	RETURN	TO
ASSERT	DOWNTO	GOTO	NOT	RSCAN	UNSIGNED
BEGIN	DROP	IF	OF	SCAN	UNTIL
BY	ELSE	INT	OR	STACK	USE
CALL	END	INTERRUPT	OTHERWISE	STORE	VARIABLE
CALLABLE	ENTRY	LABEL	PRIV	STRING	WHILE
CASE	EXTERNAL	LAND	PROC	STRUCT	XOR
CODE	FIXED	LITERAL	REAL	SUBPROC	
DEFINE	FOR	LOR	RESIDENT	THEN	

Nonreserved Keywords

The following keywords are not reserved by the compiler. You can use nonreserved keywords as identifiers anywhere identifiers are allowed, except as noted in the Restrictions column:

Keyword	Restrictions
AT	
BELOW	
BIT_FILLER	Do not use as an identifier within a structure.
BLOCK	Do not use as an identifier in a source file that contains the NAME declaration.
BYTES	Do not use as an identifier of a LITERAL or DEFINE.
C	
COBOL	
ELEMENTS	Do not use as an identifier of a LITERAL or DEFINE.
EXT	
EXTENSIBLE	
FILLER	Do not use as an identifier within a structure.
FORTRAN	
LANGUAGE	
NAME	
PASCAL	
PRIVATE	Do not use as an identifier in a source file that contains the NAME declaration.
UNSPECIFIED	
WORDS	Do not use as an identifier of a LITERAL or DEFINE.

Identifiers

Identifiers are names you declare for objects such as variables, LITERALs, DEFINEs, and procedures (including functions). Identifiers must conform to the following rules:

- ☐ They can be up to 31 characters long.
- ☐ They can begin with an alphabetic character, an underscore (_), or a circumflex (^).
- ☐ They can contain alphabetic characters, numeric characters, underscores, or circumflexes.
- ☐ They can contain lowercase and uppercase characters. The compiler treats them all as uppercase.
- ☐ They cannot be reserved keywords.
- ☐ They can be nonreserved keywords.

To separate words in identifiers, use underscores rather than circumflexes. International character-set standards allow the character printed for the circumflex to vary with each country.

Do not end identifiers with an underscore. The trailing underscore is reserved for identifiers supplied by the operating system.

The following identifiers are correct:

```
a2
TANDEM
_23456789012_00
name_with_exactly_31_characters
```

The following identifiers are incorrect:

```
2abc                                  !Begins with number
ab%99                                 !% symbol not allowed
Variable                              !Reserved word
This_name_is_too_long_so_it_is_invalid  !Too long
```

Though allowed as TAL identifiers, avoid identifiers such as:

```
Name^Using^Circumflexes
Name_Using_Trailing_Underscore_
```

Identifier Classes

Each identifier is a member of one of the following identifier classes. The compiler determines the identifier class based on how you declare the identifier. The compiler stores the identifier information in the symbol table.

Class	Meaning
Block	Global data block
Code	Read-only (P-relative) array
Variable	Simple variable, array, simple pointer, structure pointer, structure, or structure data item
DEFINE	Named text
Function	Procedure or subprocedure that returns a value
Label	Statement label
LITERAL	Named constant
PROC	Procedure or subprocedure that does not return a value
Register	Index register—R5, R6, or R7
Template	Template structure

Data Types

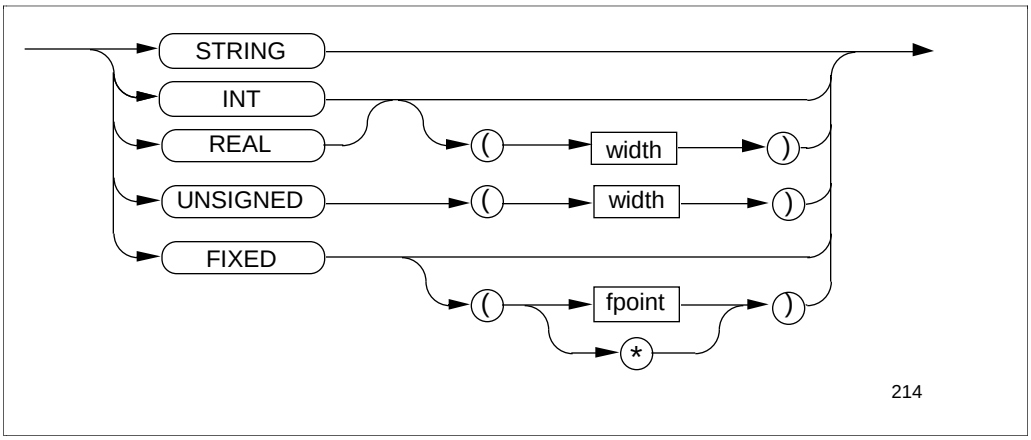
When you declare most kinds of variables, you specify a data type. The data type determines:

- ☐ The kind of values the variable can represent
- ☐ The amount of storage the compiler allocates for the variable
- ☐ The operations you can perform on the variable
- ☐ The byte or word addressing mode of the variable

Data Type	Storage Unit	Kind of Values the Data Type Can Represent
STRING	Byte	An ASCII character. An 8-bit integer in the range 0 through 255 unsigned.
INT	Word	One or two ASCII characters. A 16-bit integer in the range 0 through 65,535 unsigned or –32,768 through 32,767 signed. A standard (16-bit) address.
INT(32)	Doubleword	A 32-bit integer in the range –2,147,483,648 through +2,147,483,647. An extended (32-bit) address.
UNSIGNED	*n*-bit field	UNSIGNED(1–15) and UNSIGNED(17–31) can represent a positive unsigned integer in the range 0 through $(2^n - 1)$. UNSIGNED(16) can represent an integer in the range 0 through 65,535 unsigned or –32,768 through 32,767 signed; it can also represent a standard address
FIXED	Quadrupleword	A 64-bit fixed-point number. For FIXED(0) and FIXED(*), the range is –9,223,372,036,854,775,808 through +9,223,372,036,854,775,807.
REAL	Doubleword	A 32-bit floating-point number in the range ±8.6361685550944446E–78 through ±1.15792089237316189E77 precise to approximately seven significant digits.
REAL(64)	Quadrupleword	A 64-bit floating-point number in the same range as data type REAL, precise to approximately 17 significant digits.

For an UNSIGNED simple variable, the bit field can be 1 to 31 bits wide.
For an UNSIGNED array, the element bit field can be 1, 2, 4, or 8 bits wide.

Format of Data Types The format for specifying data types in declarations is:



width

is a constant expression that, as of the D20 release, can include LITERALs and DEFINEs. The result of the constant expression must be one of these values:

Data Type Prefix	*width,* in bits
INT	16, 32, or 64
REAL	32 or 64
UNSIGNED—simple variable, parameter, or function result	A value in the range 1 through 31
UNSIGNED—array element	1, 2, 4, or 8

fpoint

is an integer in the range –19 through 19. The default *fpoint* is 0 (no decimal places). A positive *fpoint* specifies the number of decimal places to the right of the decimal point.

```
FIXED(3)  x := 0.642F;          !Stored as 642
```

A negative *fpoint* specifies the number of integer places to the left of the decimal point.

```
FIXED(-3) y := 642945F;         !Stored as 642; accessed
                                ! as 642000
```

* (asterisk)

is a FIXED data type notation as of the D20 release. The asterisk prevents scaling of the initialization value.

Data Type Aliases The compiler accepts the following aliases for the listed data types:

Data Type	Alias
INT	INT(16)
REAL	REAL(32)
FIXED(0)	INT(64)

Storage Units Storage units are the containers in which you can access data stored in memory. The system fetches and stores all data in 16-bit words, but you can access data as any of the following storage units:

Storage Unit	Number of Bits	Data Type	Description
Byte	8	STRING	One of two bytes that make up a word
Word	16	INT	Two bytes, with byte 0 (most significant) on the left and byte 1 (least significant) on the right
Doubleword	32	INT(32), REAL	Two contiguous words
Quadrupleword	64	REAL(64), FIXED	Four contiguous words
Bit field	1–16	UNSIGNED	Contiguous bit fields within a word
Bit field	17–31	UNSIGNED	Contiguous bit fields within a doubleword

In TAL, a word is 16 bits regardless of the word size used by the system hardware.

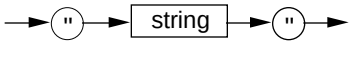
Constants

The following syntax diagrams describe:

- ☐ Character string constants (all data types)
- ☐ STRING numeric constants
- ☐ INT numeric constants
- ☐ INT(32) numeric constants
- ☐ FIXED numeric constants
- ☐ REAL and REAL(64) numeric constants
- ☐ Constant lists

Character String Constants

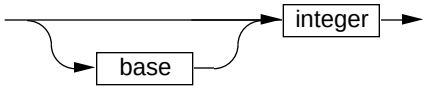
A character string constant consists of one or more ASCII characters stored in a contiguous group of bytes.



001

STRING Numeric Constants

A STRING numeric constant consists of an unsigned 8-bit integer.



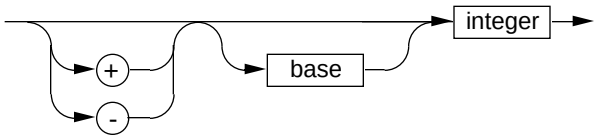
002

base

Octal	%
Binary	%B
Hexadecimal	%H

INT Numeric Constants

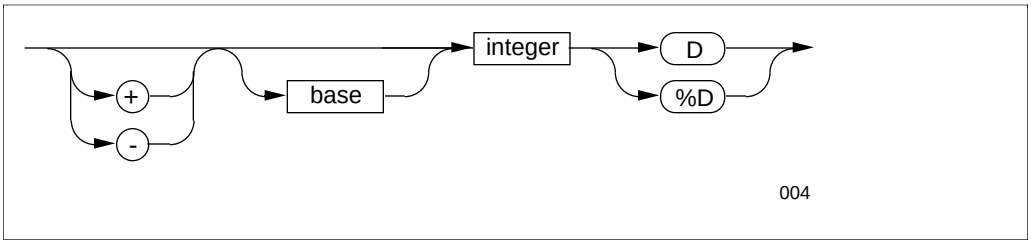
An INT numeric constant is a signed or unsigned 16-bit integer.



003

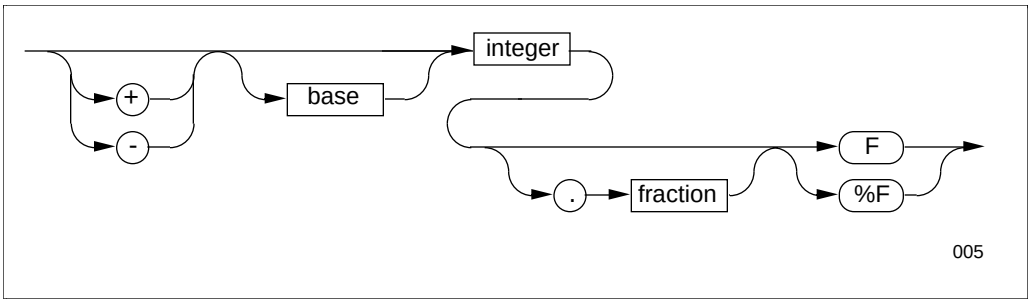
INT(32) Numeric Constants

An INT(32) numeric constant is a signed or unsigned 32-bit integer.



FIXED Numeric Constants

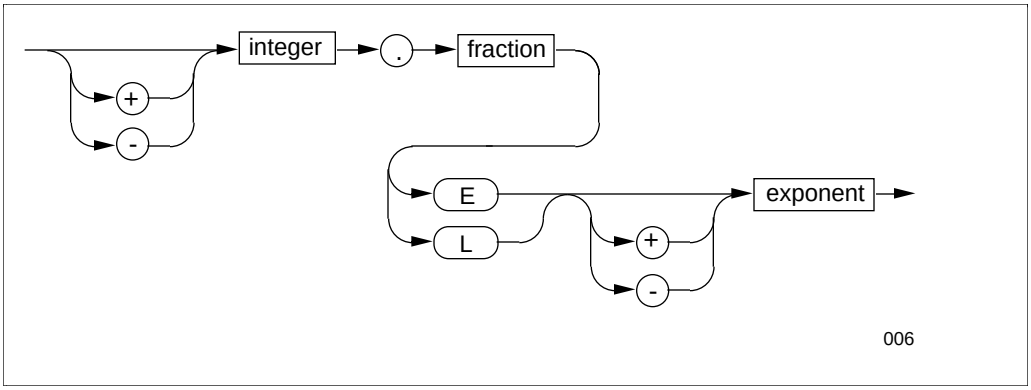
A FIXED numeric constant is a signed 64-bit fixed-point integer.



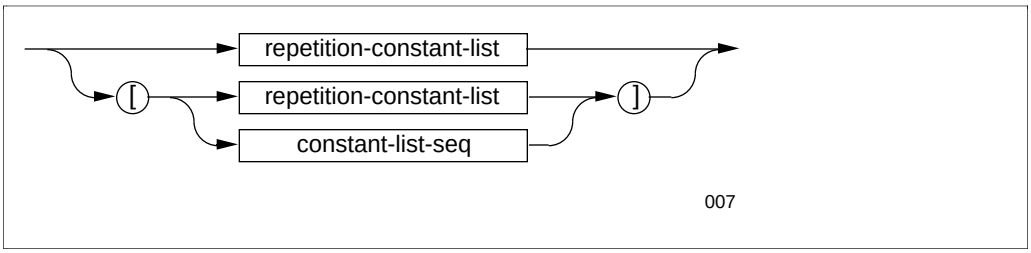
REAL and REAL(64) Numeric Constants

A REAL numeric constant is a signed 32-bit floating-point number that is precise to approximately 7 significant digits.

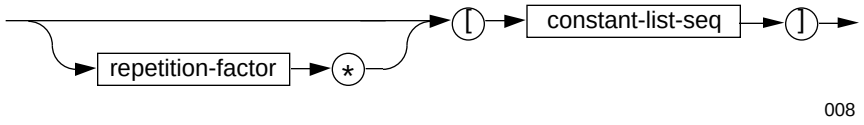
A REAL(64) numeric constant is a signed 64-bit floating-point number that is precise to approximately 17 significant digits.



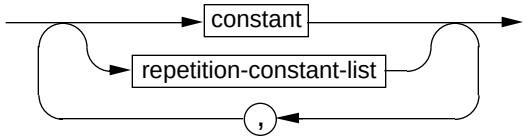
Constant Lists A constant list is a list of one or more constants.



repetition-constant-list



constant-list-seq



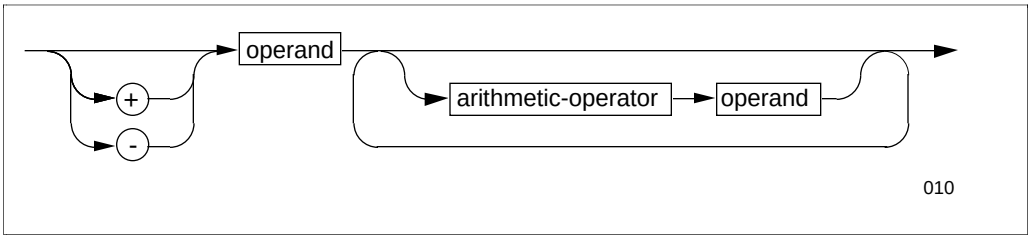
Expressions

The following syntax diagrams describe:

- ☐ Arithmetic expressions
- ☐ Conditional expressions
- ☐ Assignment expressions
- ☐ CASE expressions
- ☐ IF expressions
- ☐ Group comparison expressions
- ☐ Bit extractions
- ☐ Bit shifts

Arithmetic Expressions

An arithmetic expression is a sequence of operands and arithmetic operators that computes a single numeric value of a specific data type.



operand

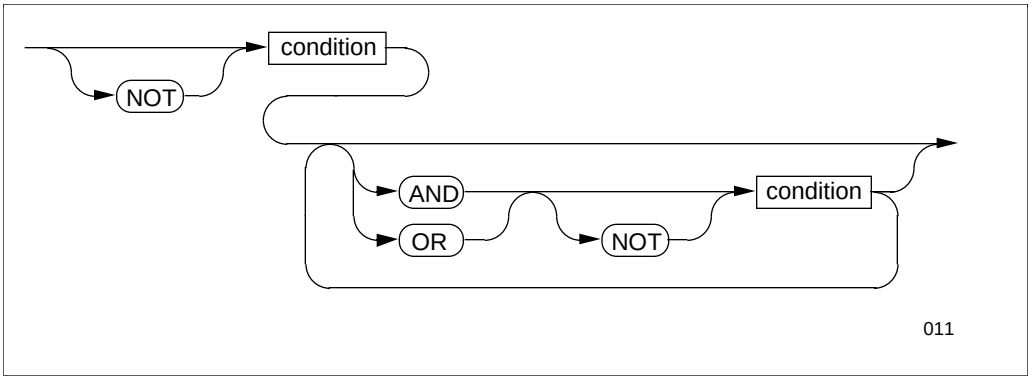
Variable	The identifier of a simple variable, array element, pointer, structure data item, or equivalenced variable, with or without @ or an index
Constant	A character string or numeric constant
LITERAL	The identifier of a named constant
Function invocation	The invocation of a procedure that returns a value
(expression)	Any expression, enclosed in parentheses
Code space item	The identifier of a procedure, subprocedure, or label prefixed with @ or a read-only array optionally prefixed with @, with or without an index

arithmetic-operator

Signed arithmetic operator:	`+, -, *, /`
Unsigned arithmetic operator:	`'+', '-', '*', '/', '\'`
Logical operator:	`LOR, LAND, XOR`

Conditional Expressions

A conditional expression is a sequence of conditions and Boolean or relational operators that establishes the relationship between values.

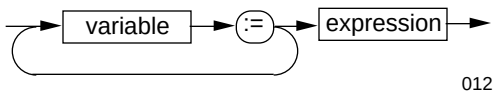


condition

Relational expression	Two conditions connected by a relational operator. The result type is INT; a –1 if true or a 0 if false.
Group comparison expression	Unsigned comparison of a group of contiguous elements with another. The result type is INT; a –1 if true or a 0 if false.
(conditional expression)	A conditional expression enclosed in parentheses. The result type is INT; a –1 if true or a 0 if false.
Arithmetic expression	An arithmetic, assignment, CASE, or IF expression that has an INT result. The expression is treated as true if its value is not 0 and false if its value is 0.
Relational operator	A signed or unsigned relational operator that tests a condition code. Condition code settings are CCL (negative), CCE (0), or CCG (positive).

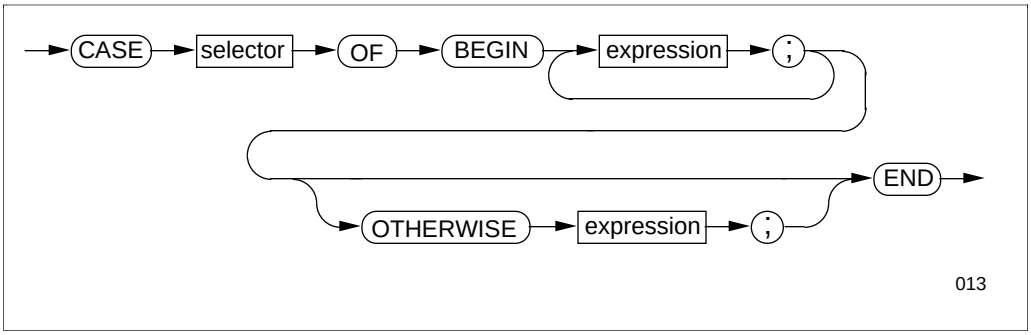
Assignment Expressions

The assignment expression assigns the value of an expression to a variable.



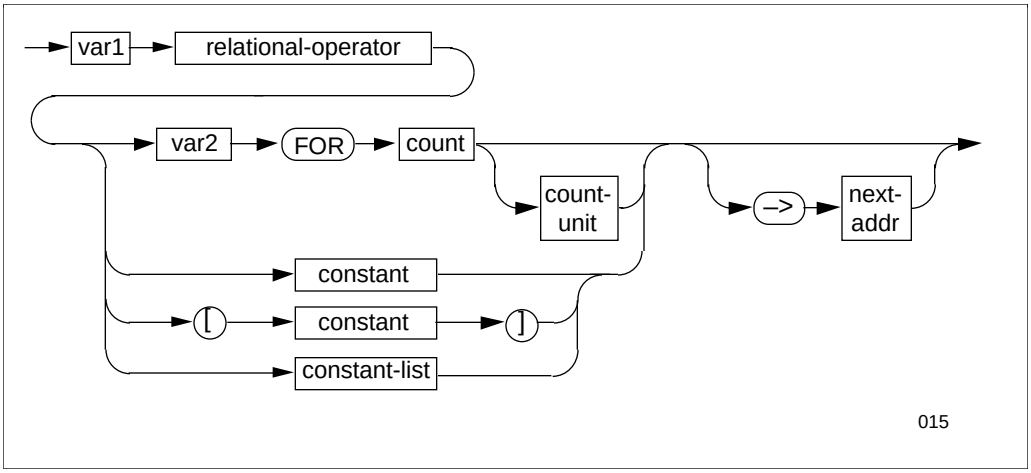
CASE Expressions

The CASE expression selects one of several expressions.



Group Comparison Expressions

The group comparison expression compares a variable with a variable or a constant.



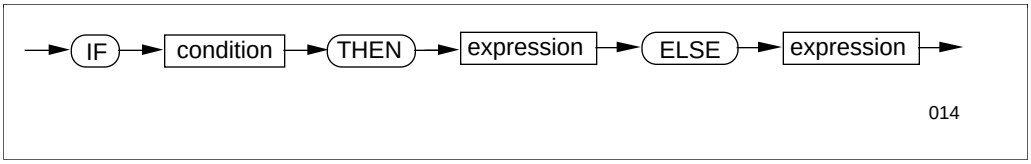
relational-operator

Signed relational operator: `<, =, >, <=, >=, <>`
Unsigned relational operator: `'<', '=', '>', '<=', '>=', '<>'`

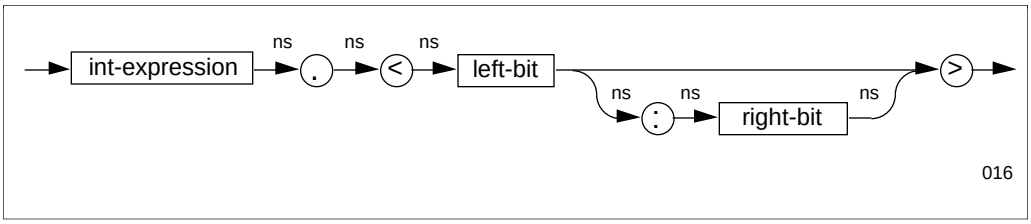
count-unit

BYTES	Compares *count* bytes. If *var1* and *var2* both have word addresses, BYTES generates a word comparison for (*count* + 1) / 2 words.
WORDS	Compares *count* words.
ELEMENTS	Compares *count* occurrences of structures. structure pointers, and substructures. Otherwise, compares *count* units depending on data type.

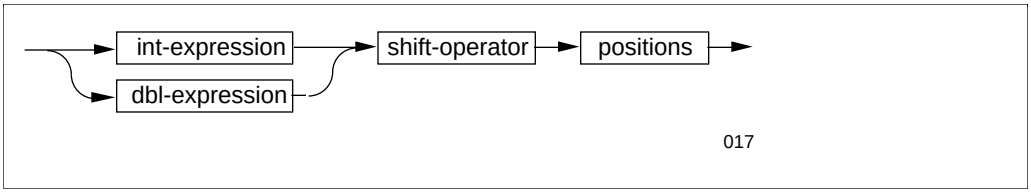
IF Expressions The IF expression conditionally selects one of two expressions, usually for assignment to a variable.



Bit Extractions A bit extraction accesses a bit field in an INT expression without altering the expression.



Bit Shifts A bit shift operation shifts a bit field a specified number of positions to the left or to the right within a variable without altering the variable.



shift-operator

```
'<<', '>>', <<, or >>
```

Declarations

Declaration syntax diagrams describe:

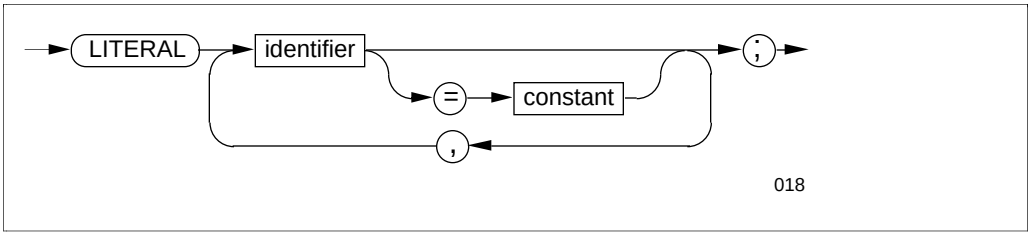
- ☐ LITERALs and DEFINEs
- ☐ Simple variables
- ☐ Arrays and read-only arrays
- ☐ Structures—definition structures, template structures, referral structures
- ☐ Structure items—simple variables, arrays, substructures, filler bytes, filler bits, simple pointers, structure pointers, and redefinitions
- ☐ Simple pointers and structure pointers
- ☐ Equivalenced variables
- ☐ NAMEs and BLOCKs
- ☐ Procedures and subprocedures

LITERAL and DEFINE Declarations

The following syntax diagrams describe LITERAL and DEFINE declarations.

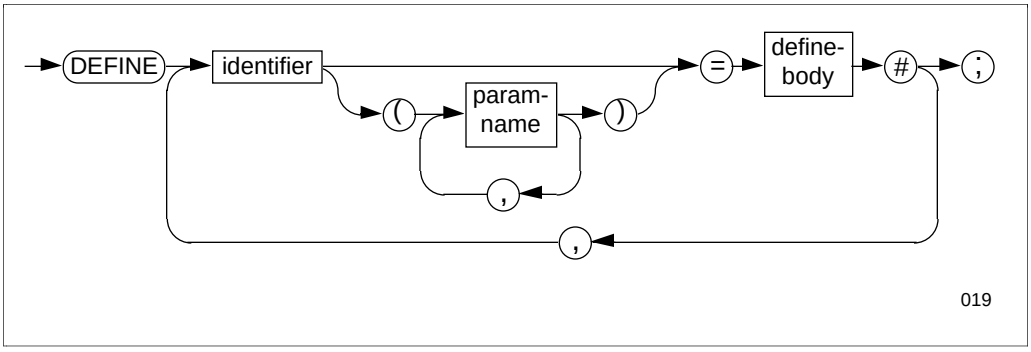
LITERALs

A LITERAL associates an identifier with a constant.



DEFINEs

A DEFINE associates an identifier (and parameters if any) with text.

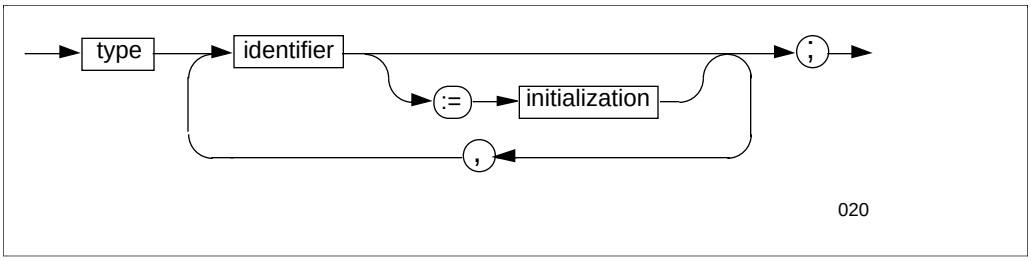


Simple Variable Declarations

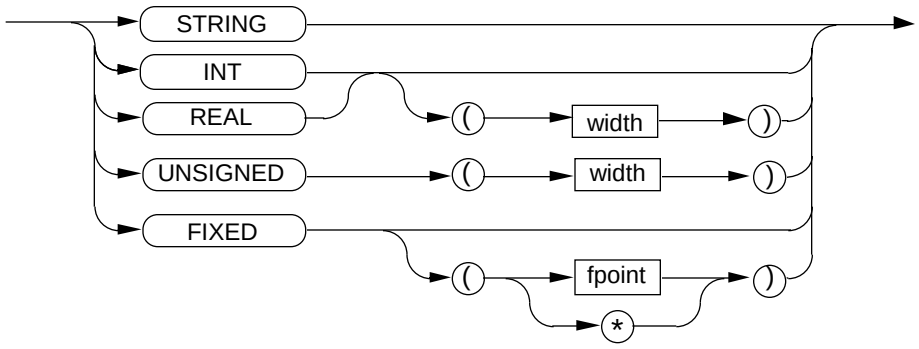
The following syntax diagram describes simple variable declarations:

Simple Variables

A simple variable associates an identifier with a single-element data item of a specified data type.



type

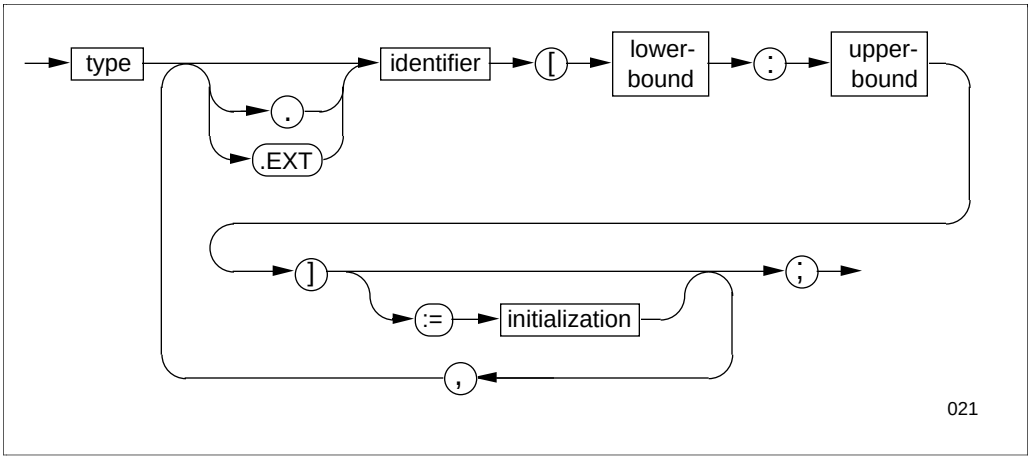


Array Declarations

The following syntax diagrams describe array and read-only array declarations:

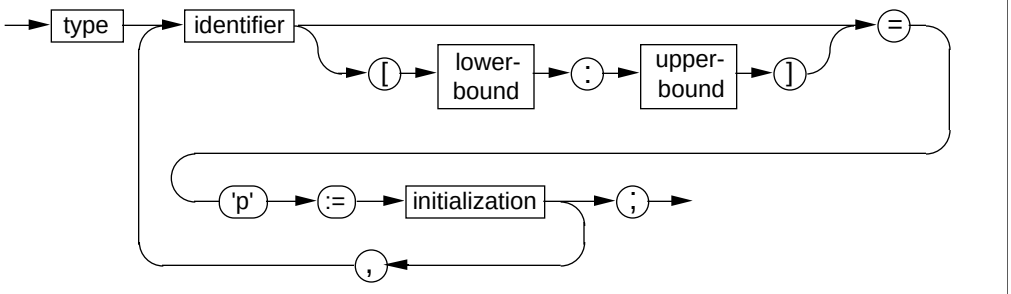
Arrays

An array associates an identifier with a one-dimensional set of elements of the same data type.



Read-Only Arrays

A read-only array associates an identifier with a one-dimensional and nonmodifiable set of elements of the same data type. A read-only array is located in a user code segment.



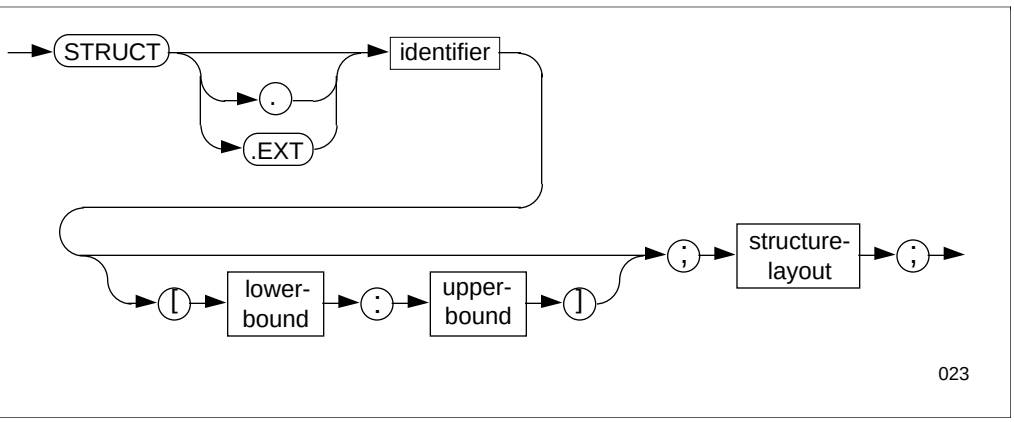
Structure Declarations

The following syntax diagrams describe:

- Structures—definition structures, template structures, referral structures
- Structure items—simple variables, arrays, substructures, filler bytes, filler bits, simple pointers, structure pointers, and redefinitions

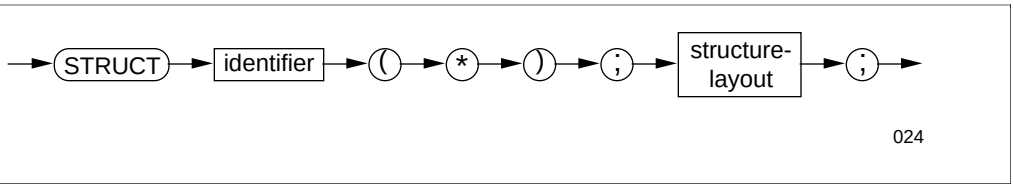
Definition Structures

A definition structure associates an identifier with a structure layout and allocates storage for it.



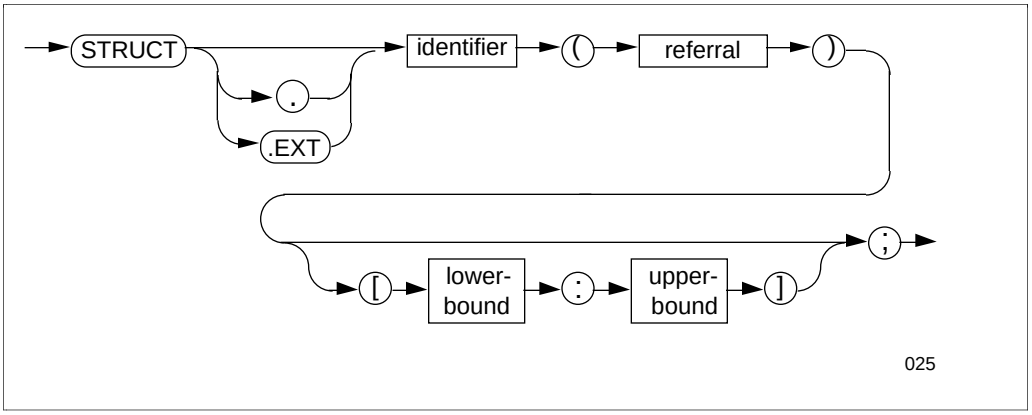
Template Structures

A template structure associates an identifier with a structure layout but allocates no storage for it.



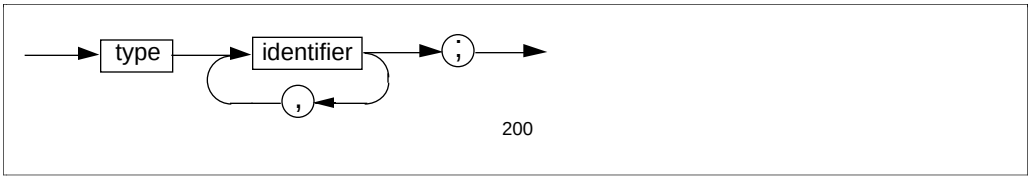
Referral Structures

A referral structure associates an identifier with a structure whose layout is the same as a previously declared structure and allocates storage for it.



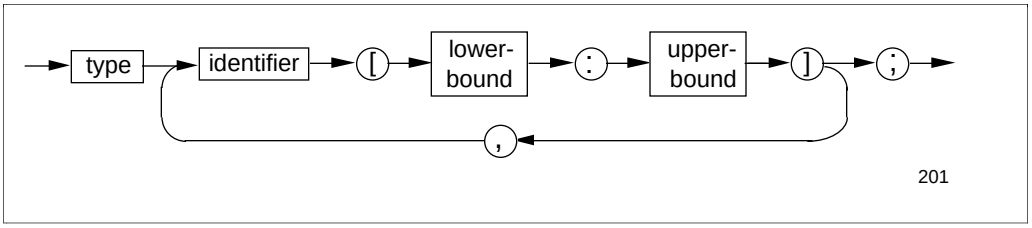
Simple Variables Declared in Structures

A simple variable can be declared inside a structure.



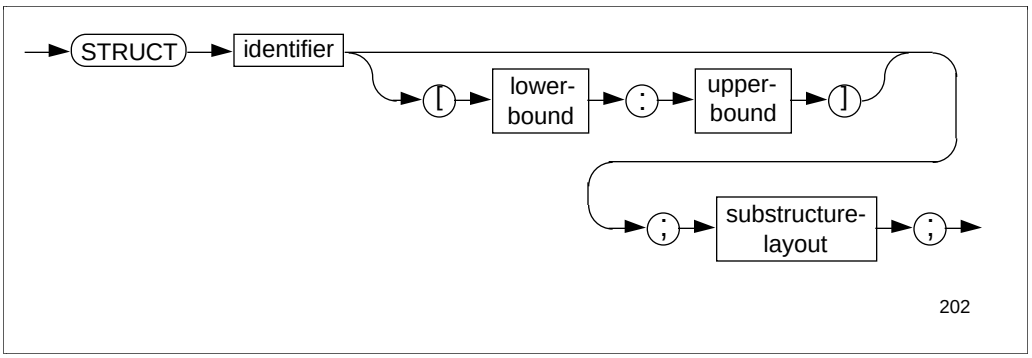
Arrays Declared in Structures

An array can be declared inside a structure.



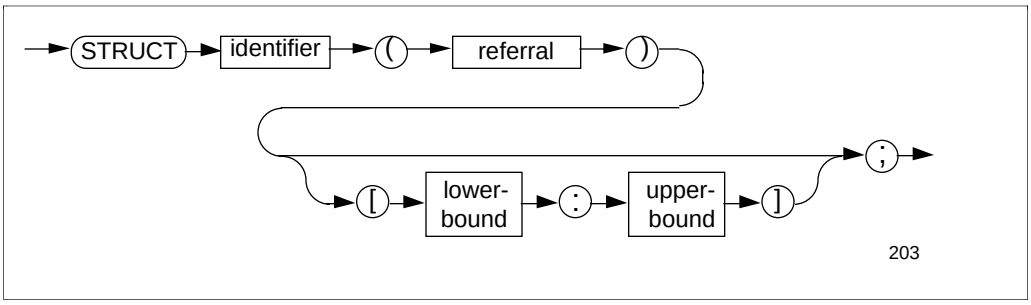
Definition Substructures

A definition substructure can be declared inside a structure.



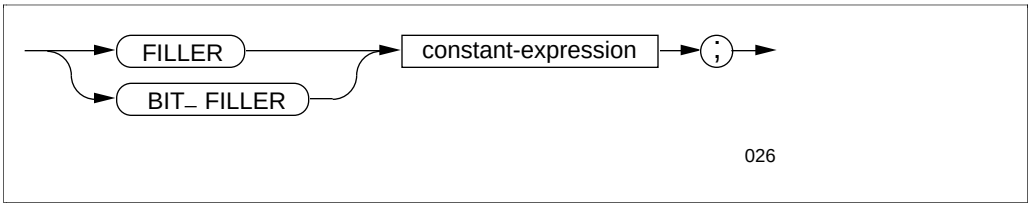
Referral Substructures

A referral substructure can be declared inside a structure.



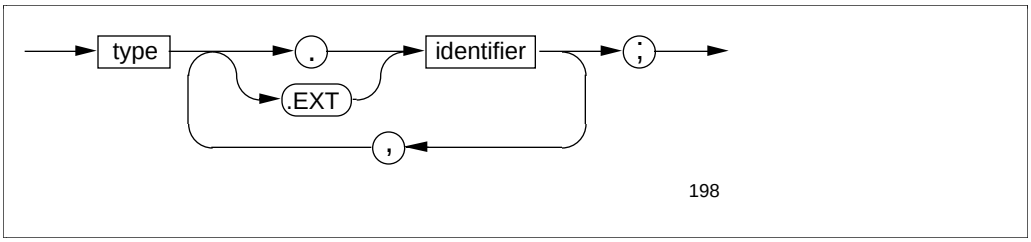
Fillers in Structures

A filler is a byte or bit place holder in a structure.



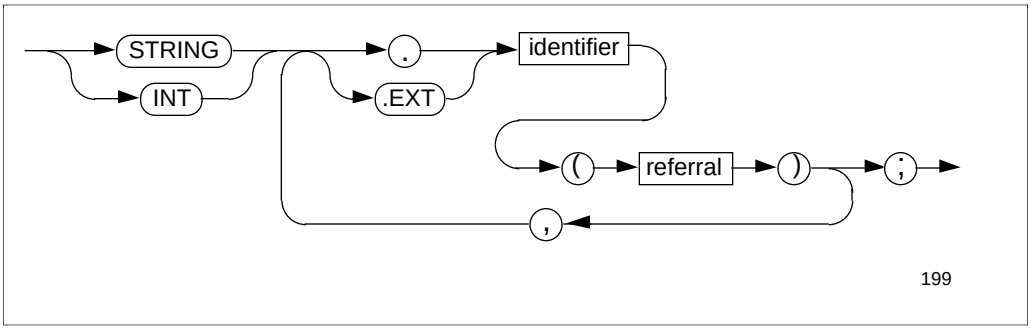
Simple Pointers Declared in Structures

A simple pointer can be declared inside a structure.



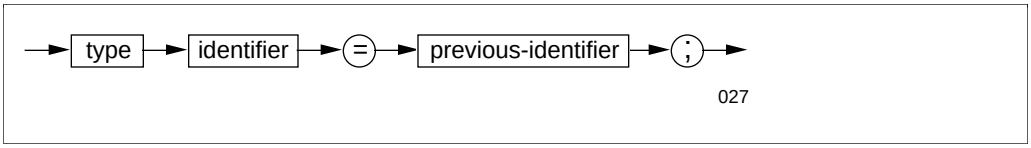
Structure Pointers Declared in Structures

A structure pointer can be declared inside a structure.



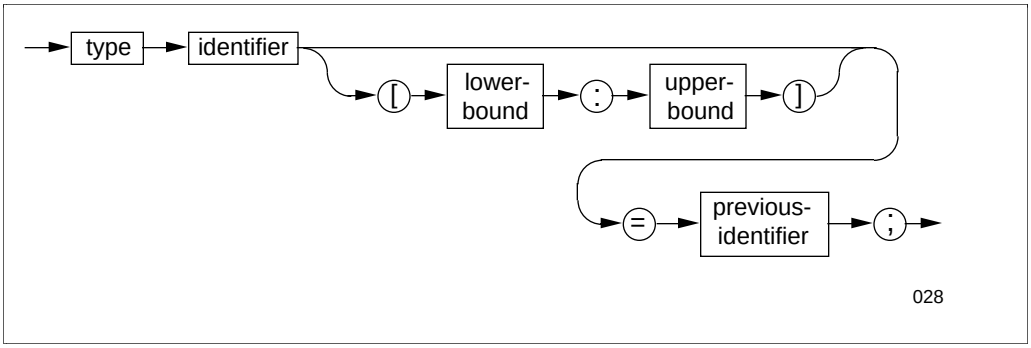
Simple Variable Redefinitions

A simple variable redefinition associates a new simple variable with a previous item at the same BEGIN-END level of a structure.



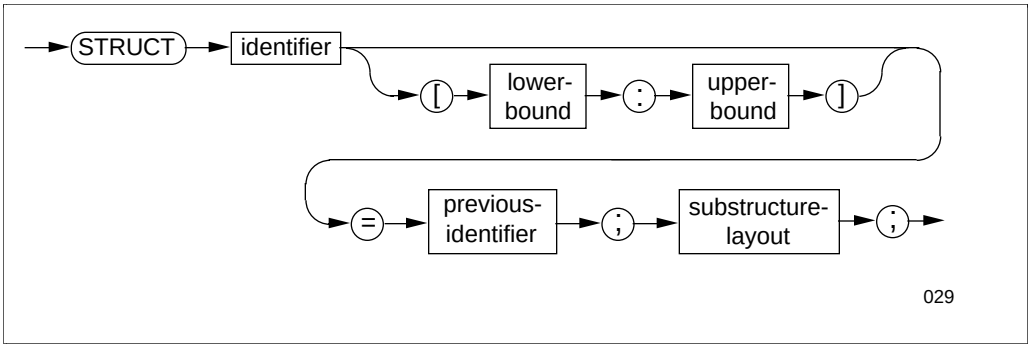
Array Redefinitions

An array redefinition associates a new array with a previous item at the same BEGIN-END level of a structure.



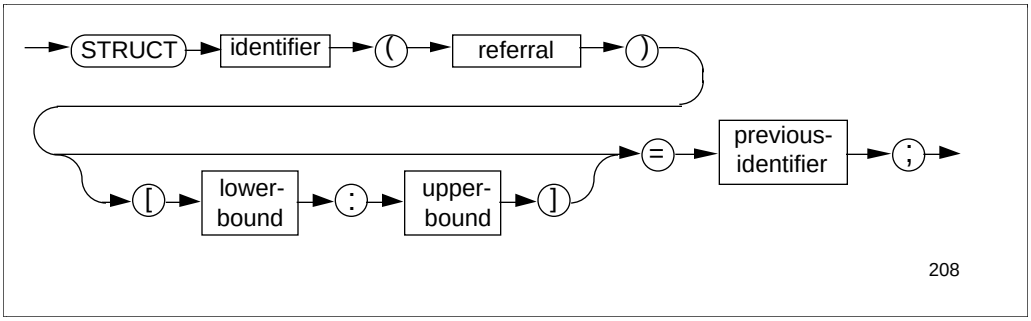
Definition Substructure Redefinitions

A definition substructure redefinition associates a definition substructure with a previous item at the same BEGIN-END level of a structure.



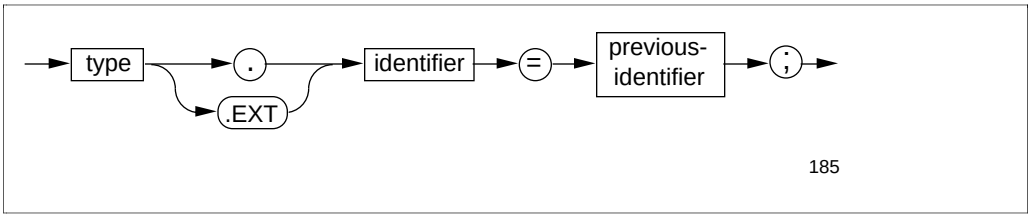
Referral Substructure Redefinitions

A referral substructure redefinition associates a referral substructure with a previous item at the same BEGIN-END level of a structure.



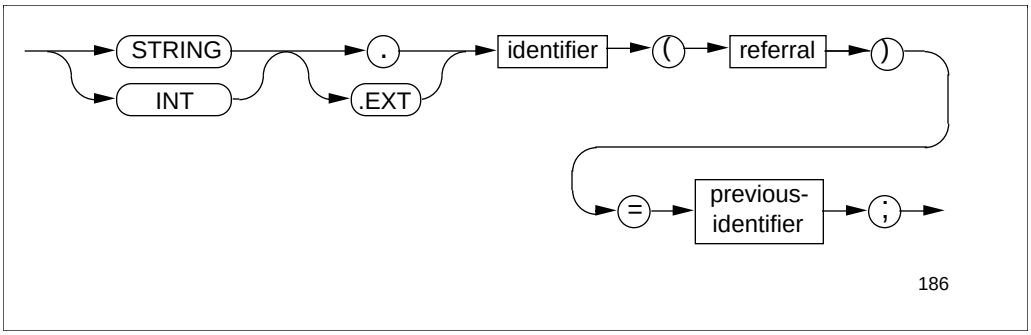
Simple Pointer Redefinitions

A simple pointer redefinition associates a new simple pointer with a previous item at the same BEGIN-END level of a structure.



Structure Pointer Redefinitions

A structure pointer redefinition associates a structure pointer with a previous item at the same BEGIN-END level of a structure.

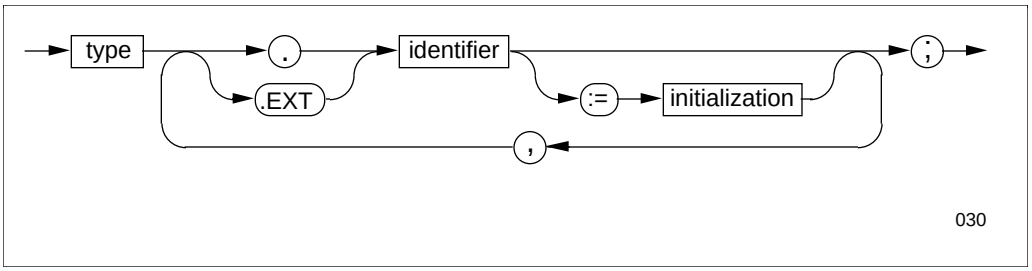


Pointer Declarations

The following syntax diagrams describe simple pointer and structure pointer declarations.

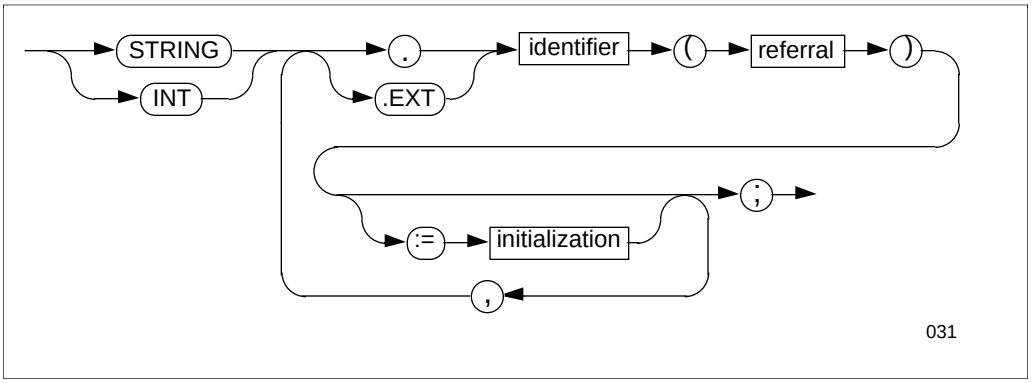
Simple Pointers

A simple pointer is a variable you load with a memory address, usually of a simple variable or array, which you access with this simple pointer.



Structure Pointers

A structure pointer is a variable you load with a memory address of a structure, which you access with this structure pointer.

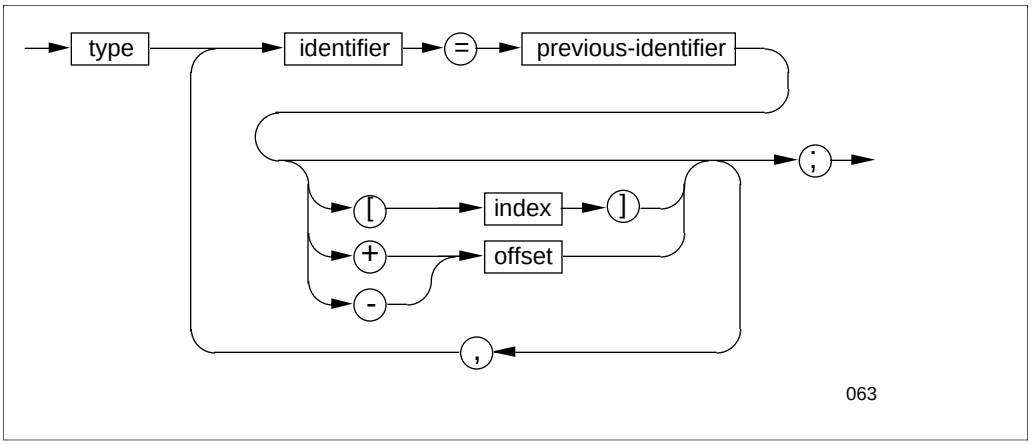


Equivalenced Variable Declarations

The following syntax diagrams describe equivalenced variable declarations for simple variables, simple pointers, structures, and structure pointers.

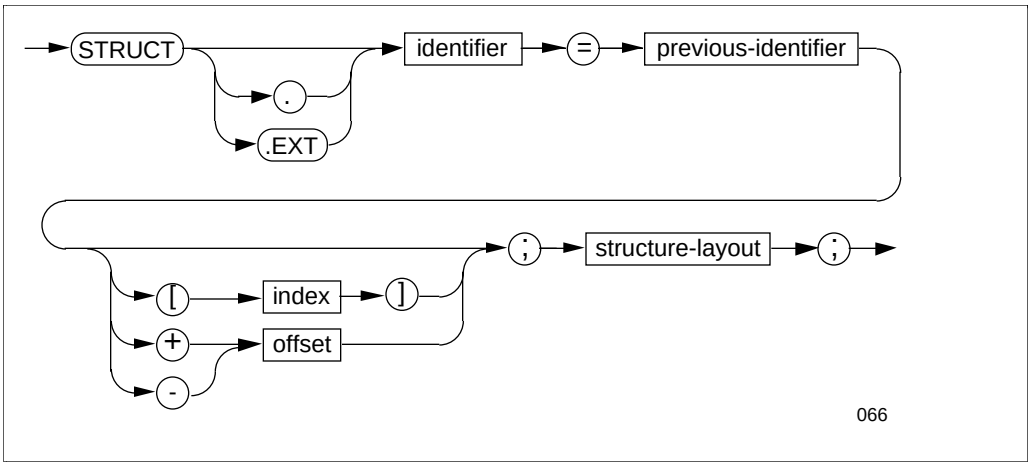
Equivalenced Simple Variables

An equivalenced simple variable associates a new simple variable with a previously declared variable.



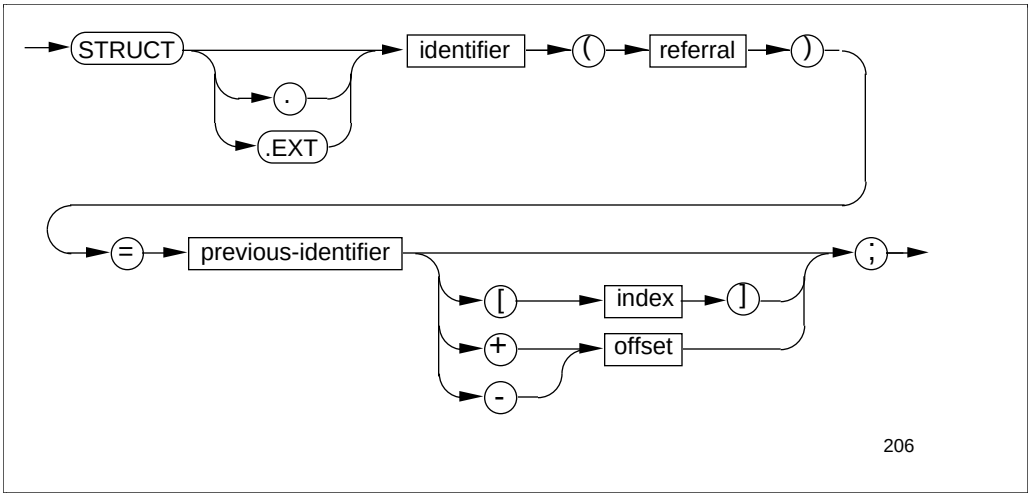
Equivalenced Definition Structures

An equivalenced definition structure associates a new definition structure with a previously declared variable.



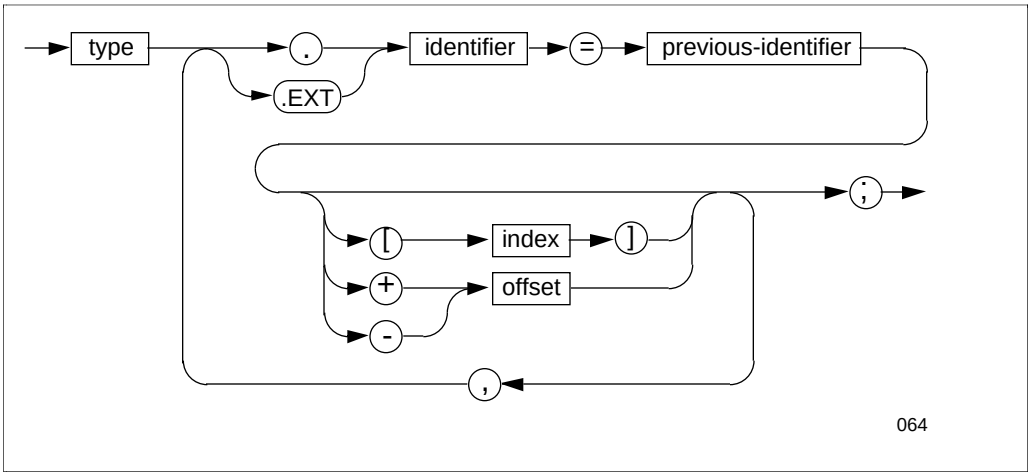
Equivalenced Referral Structures

An equivalenced referral structure associates a new referral structure with a previously declared variable.



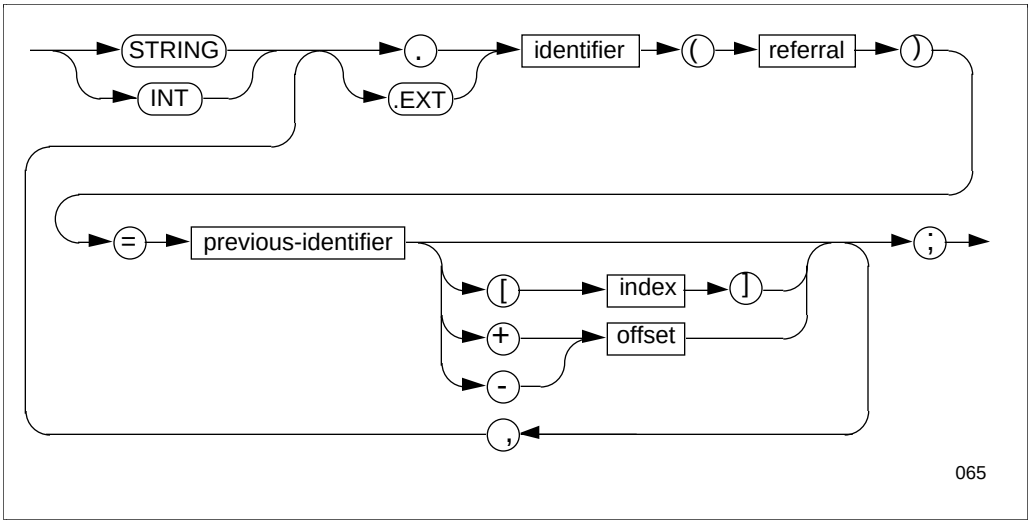
Equivalenced Simple Pointers

An equivalenced simple pointer associates a new simple pointer with a previously declared variable.



Equivalenced Structure Pointers

An equivalenced structure pointer associates a new structure pointer with a previously declared variable.

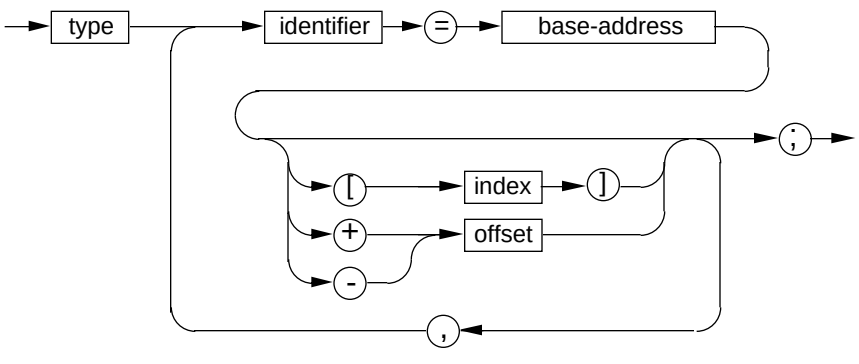


Base-Address Equivalenced Variable Declarations

The following syntax diagrams describe base-addressed equivalenced variable declarations for simple variables, simple pointers, structures, and structure pointers.

Base-Address Equivalenced Simple Variables

A base-addressed equivalenced simple variable associates a simple variable with a global, local, or top-of-stack base address.



base-address

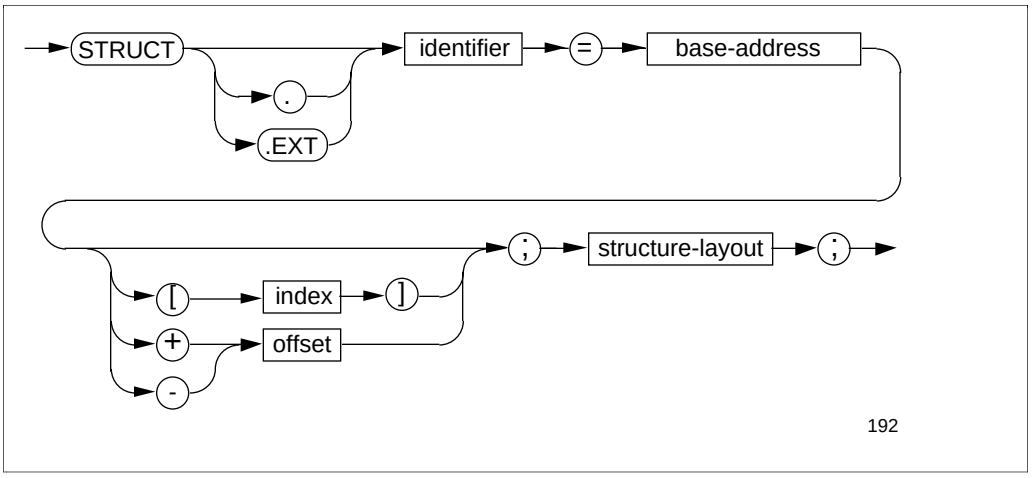
'G'	Denotes global addressing relative to G[0]
'L'	Denotes local addressing relative to L[0]
'S'	Denotes top-of-stack addressing relative to S[0]

index and offset

0 through 255	For 'G' addressing
–255 through 127	For 'L' addressing
–31 through 0	For 'S' addressing

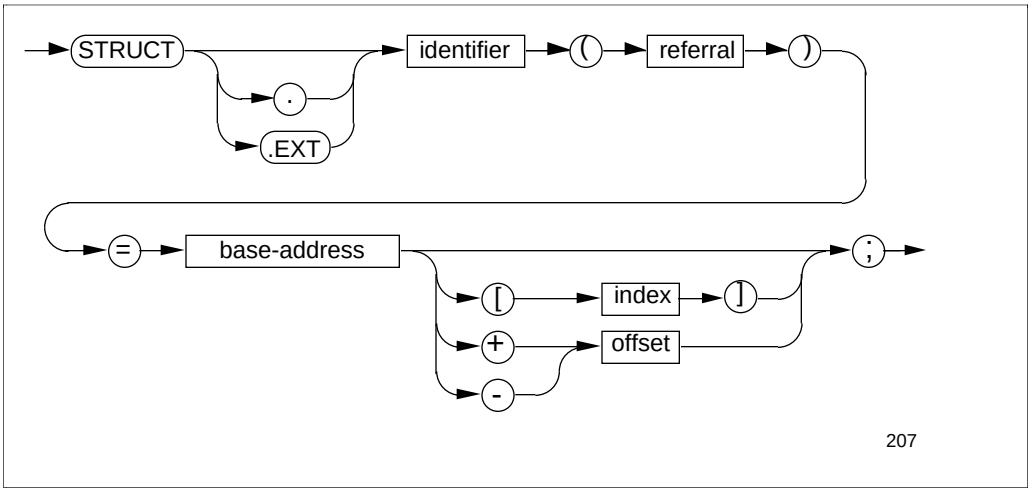
Base-Address Equivalenced Definition Structures

A base-addressed equivalenced definition structure associates a definition structure with a global, local, or top-of-stack base address.



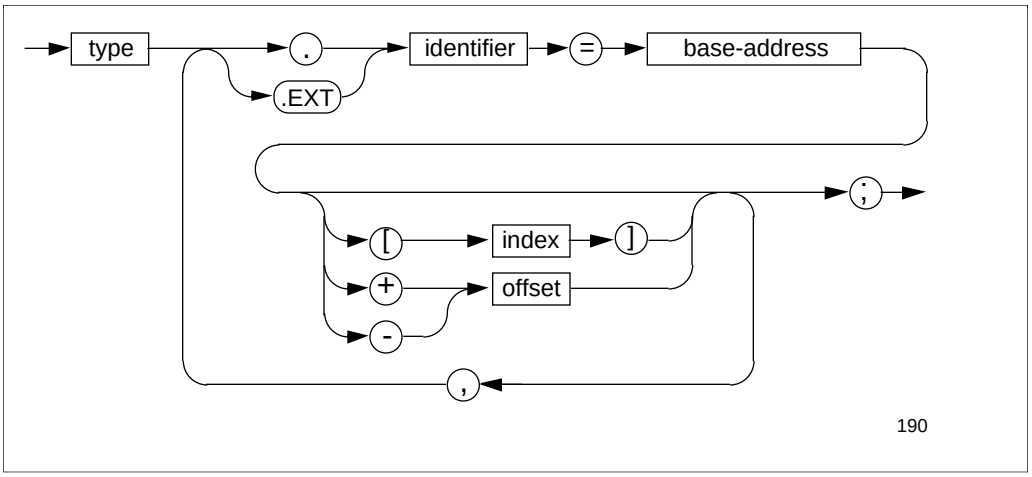
Base-Address Equivalenced Referral Structures

A base-addressed equivalenced referral structure associates a referral structure with a global, local, or top-of-stack base address.



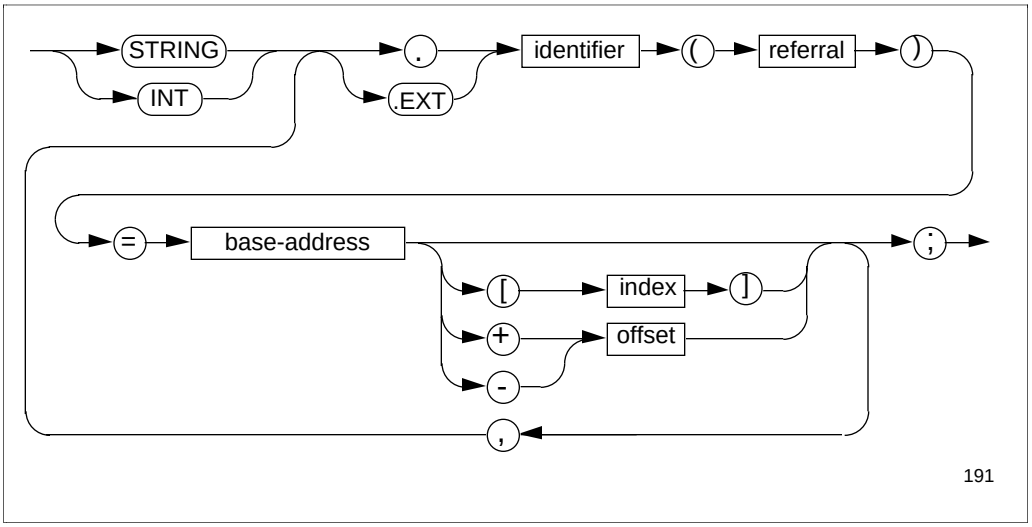
Base-Address Equivalenced Simple Pointers

A base-addressed equivalenced simple pointer associates a simple pointer with a global, local, or top-of-stack base address.



Base-Address Equivalenced Structure Pointers

A base-addressed equivalenced structure pointer associates a structure pointer with a global, local, or top-of-stack base address.

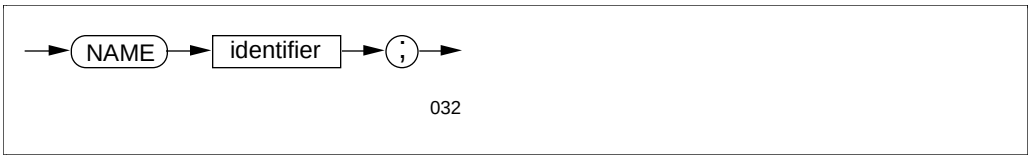


NAME and BLOCK Declarations

The following syntax diagrams describe NAME and BLOCK declarations.

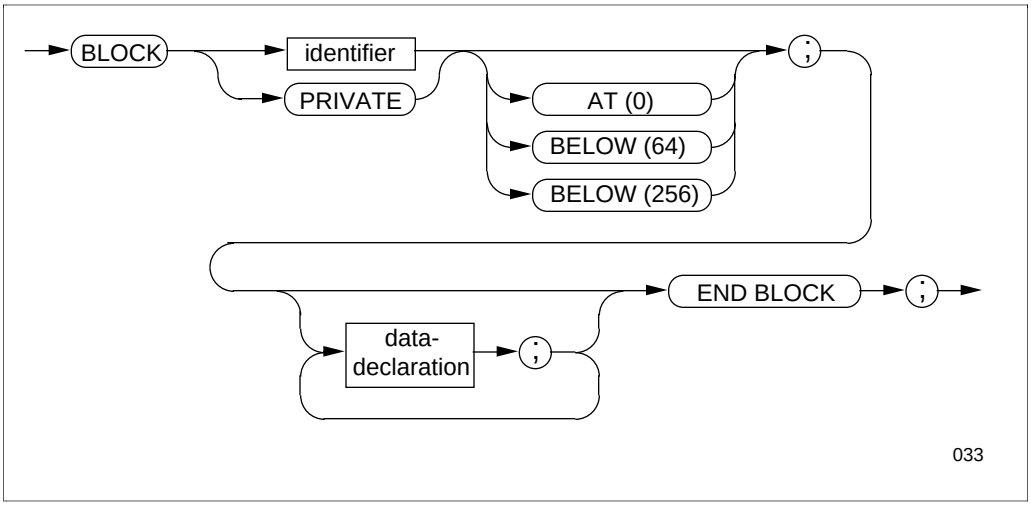
NAMEs

The NAME declaration assigns an identifier to a compilation unit and to its private global data block if it has one.



BLOCKs

The BLOCK declaration groups global data declarations into a named or private relocatable global data block.

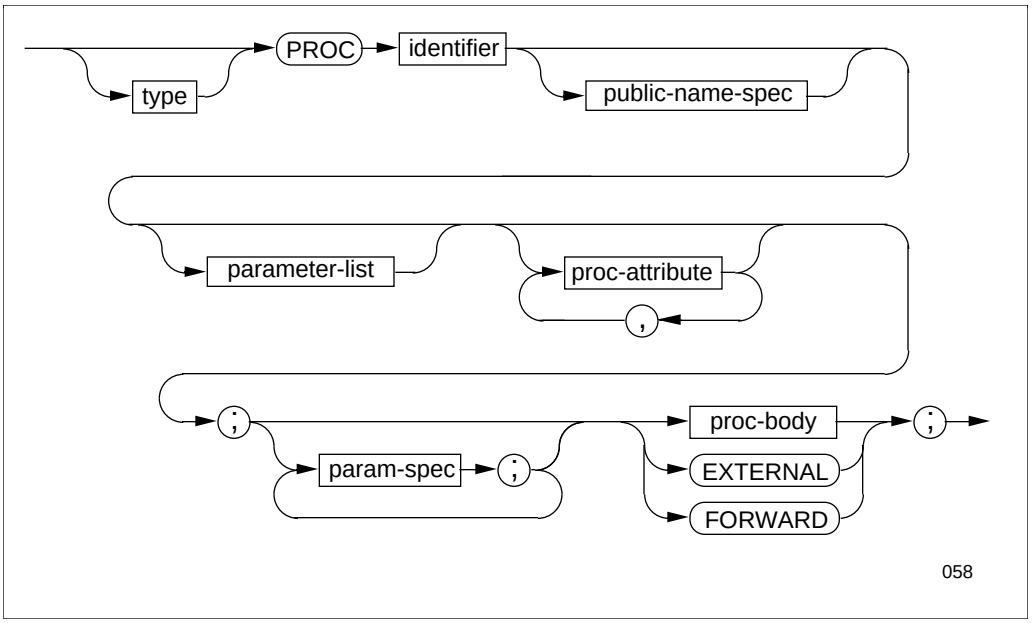


Procedure and Subprocedure Declarations

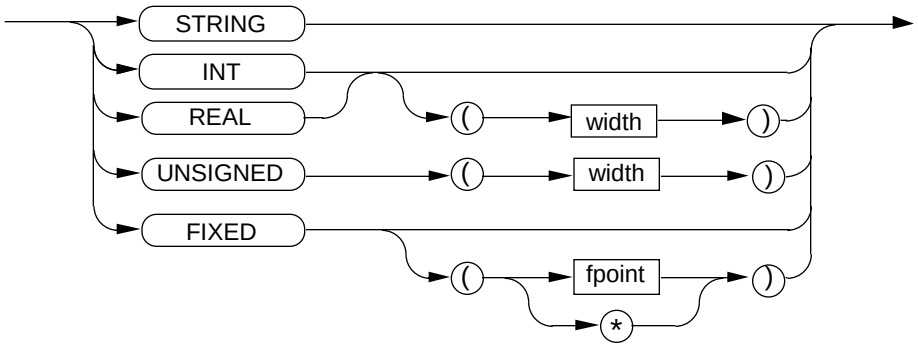
The following syntax diagrams describe procedure, subprocedure, entry-point, and label declarations.

Procedures

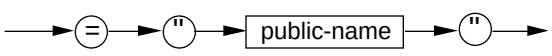
A procedure is a program unit that is callable from anywhere in the program.



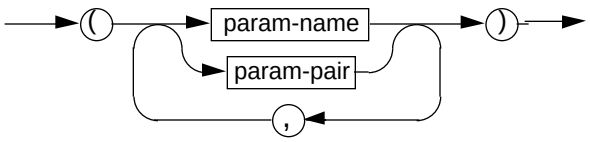
type



public-name-spec

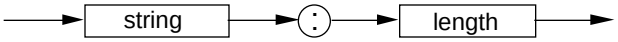


parameter-list



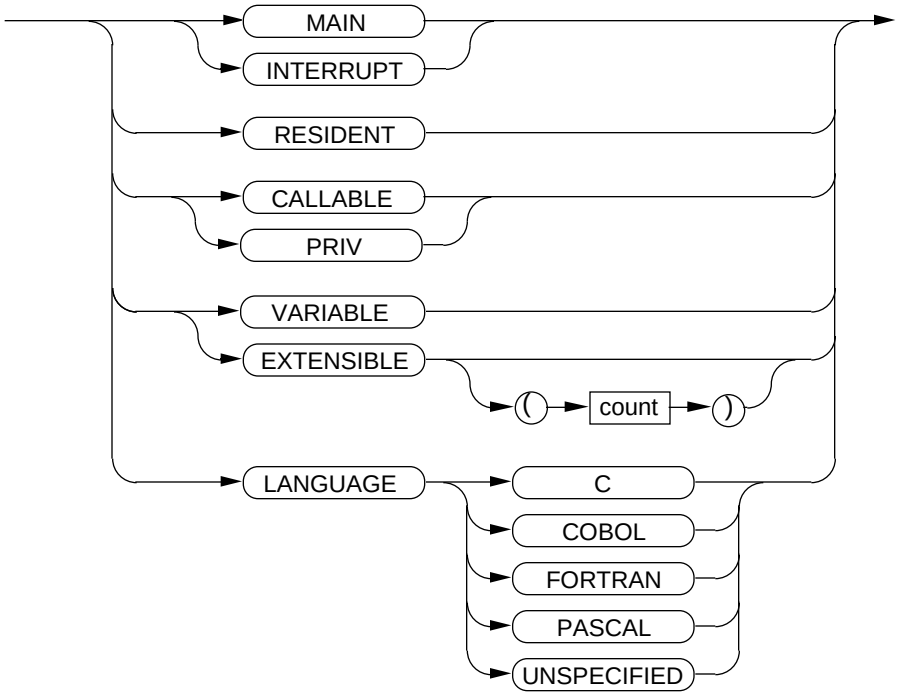
210

param-pair



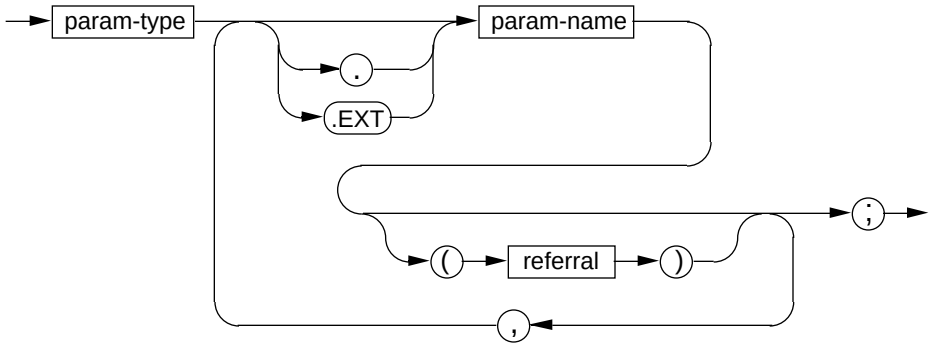
039

proc-attribute



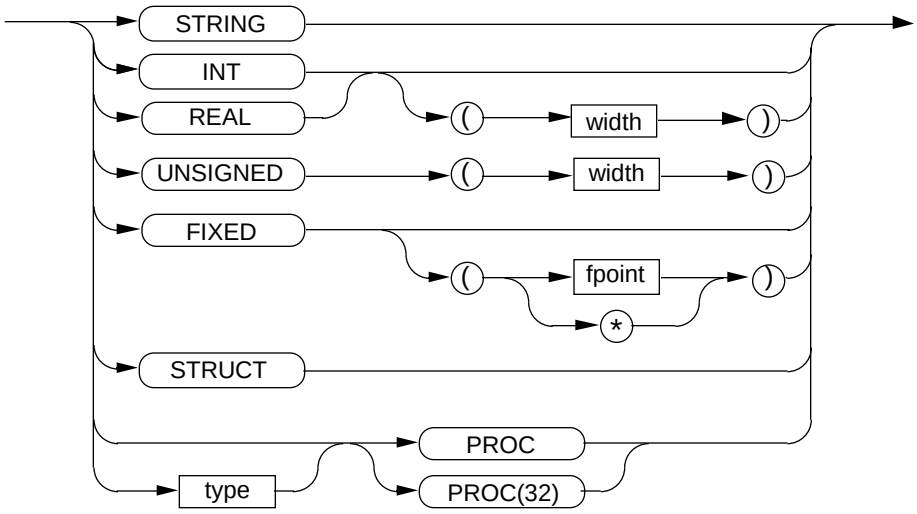
187

param-spec



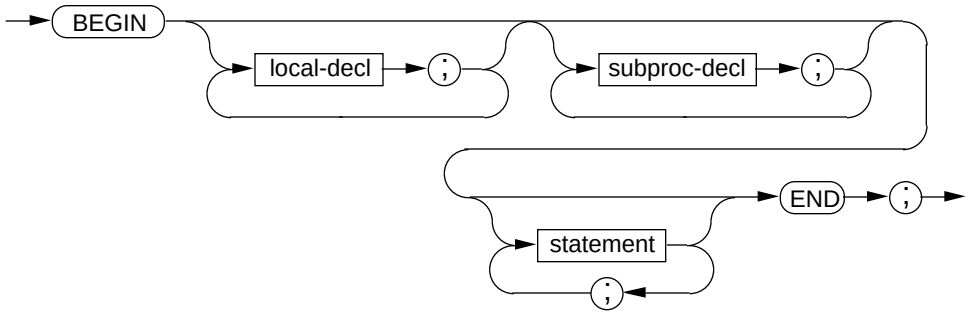
060

param-type



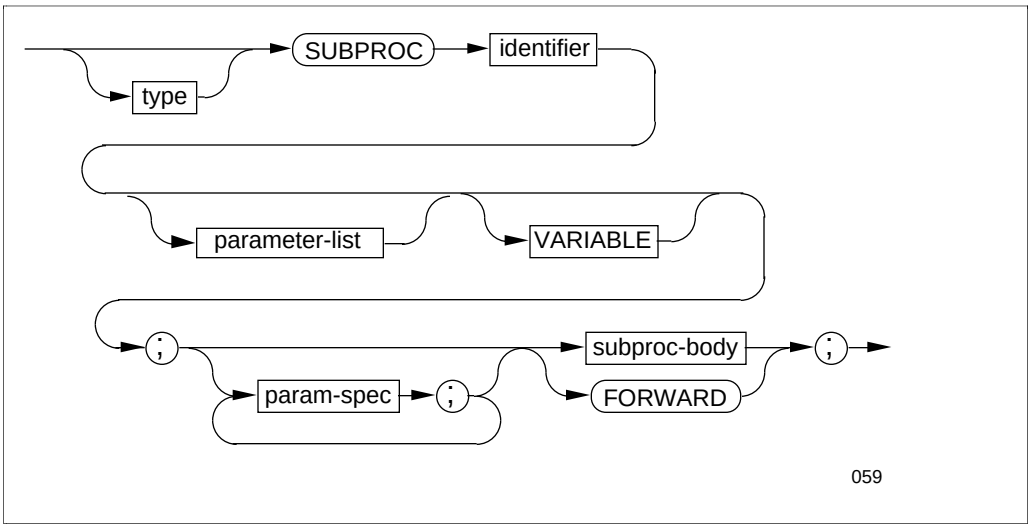
215

proc-body

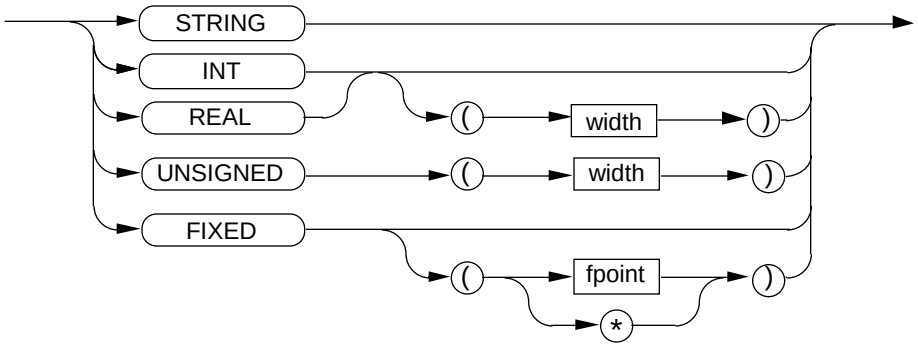


061

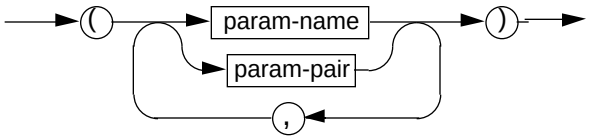
Subprocedures A subprocedure is a program unit that is callable from anywhere in the procedure.



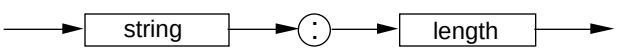
type



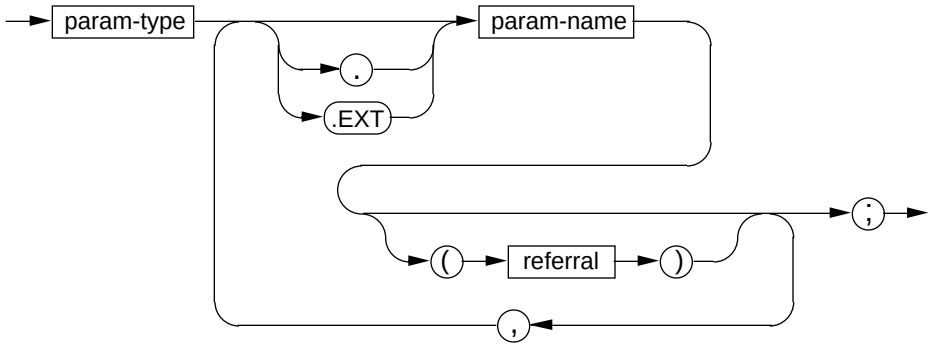
parameter-list



param-pair

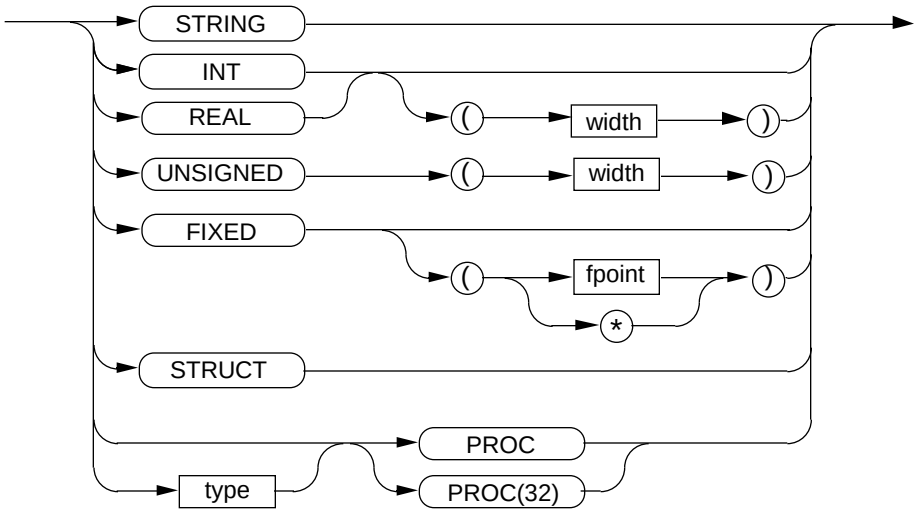


param-spec



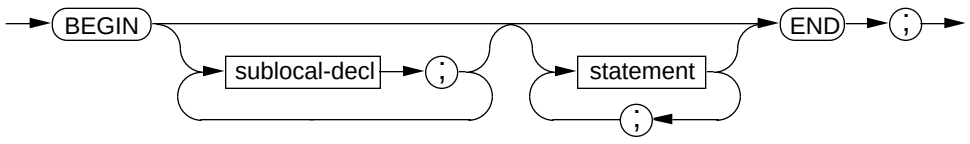
060

param-type



215

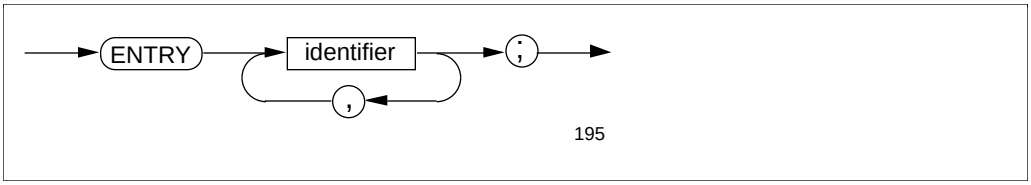
subproc-body



062

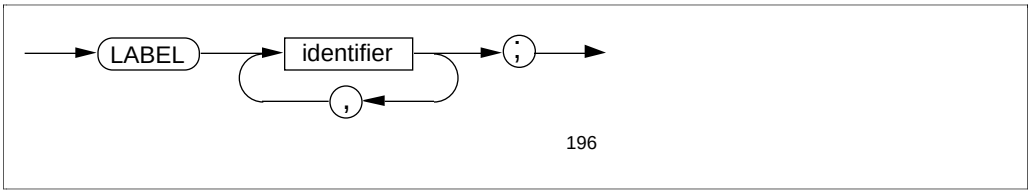
Entry Points

The entry-point declaration associates an identifier with a secondary location from which execution can start in a procedure or subprocedure.



Labels

The LABEL declaration reserves an identifier for later use as a label within the encompassing procedure or subprocedure.

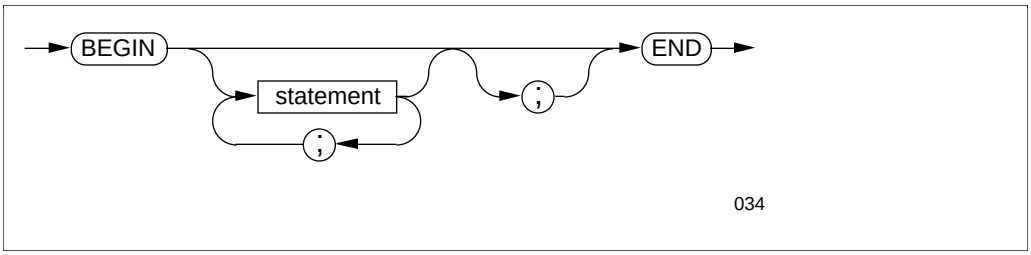


Statements

The following syntax diagrams describe statements in alphabetic order.

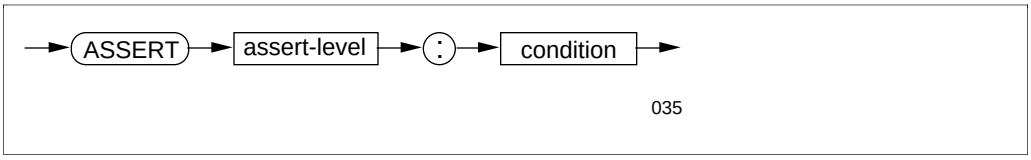
Compound Statements

A compound statement is a BEGIN-END construct that groups statements to form a single logical statement.



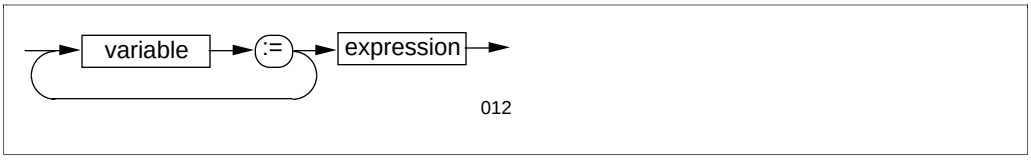
ASSERT Statement

The ASSERT statement conditionally invokes the procedure specified in an ASSERTION directive.



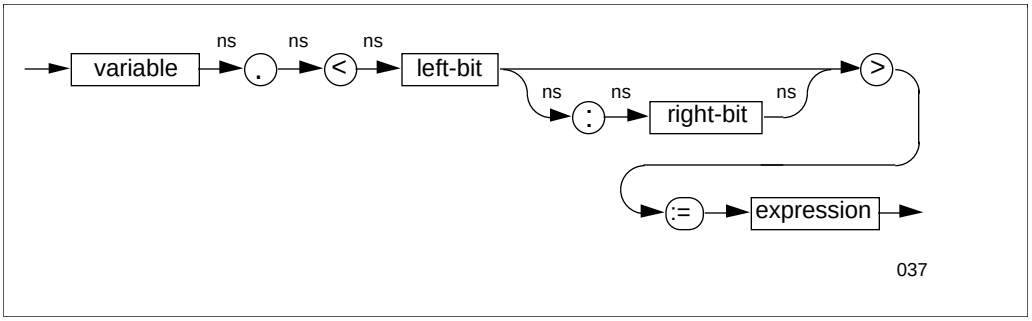
Assignment Statement

The assignment statement assigns a value to a previously declared variable.



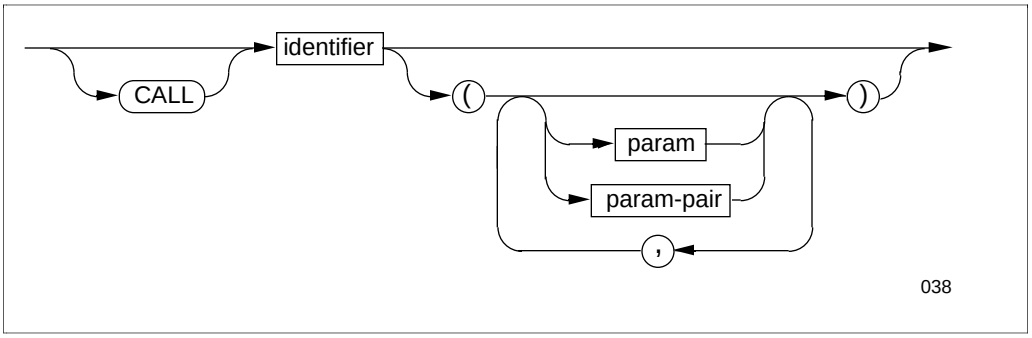
Bit Deposit Assignment Statement

The bit-deposit assignment statement assigns a value to a bit field in a variable.

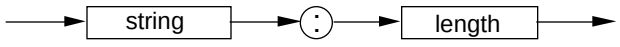


CALL Statement

The CALL statement invokes a procedure, subprocedure, or entry point, and optionally passes parameters to it.

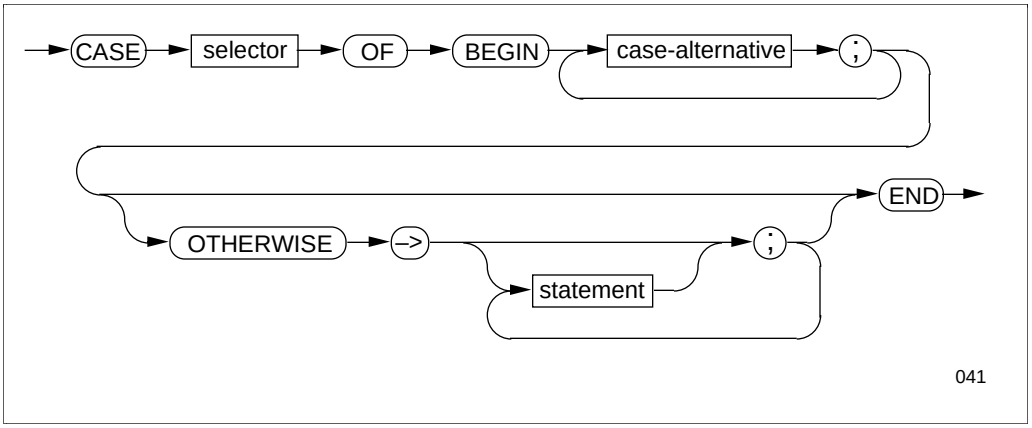


param-pair

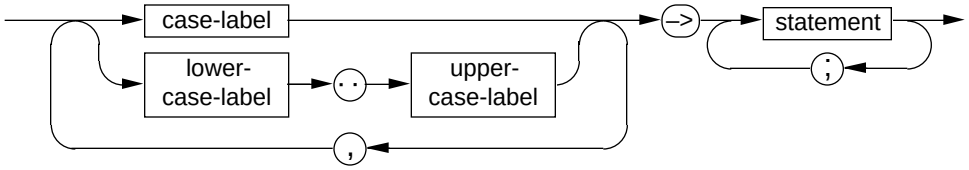


Labeled CASE Statement

The labeled CASE statement executes a choice of statements the selector value matches a case label associated with those statements.

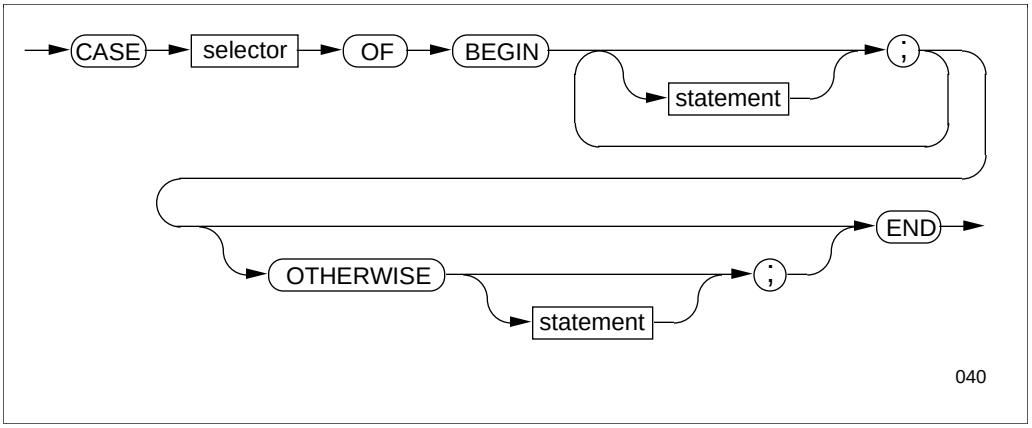


case-alternative

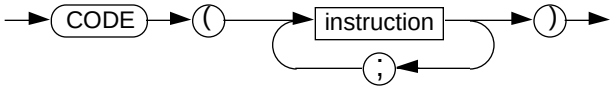


Unlabeled CASE Statement

The unlabeled CASE statement executes a choice of statements based on an inclusive range of implicit selector values, from 0 through *n*, with one statement for each value..

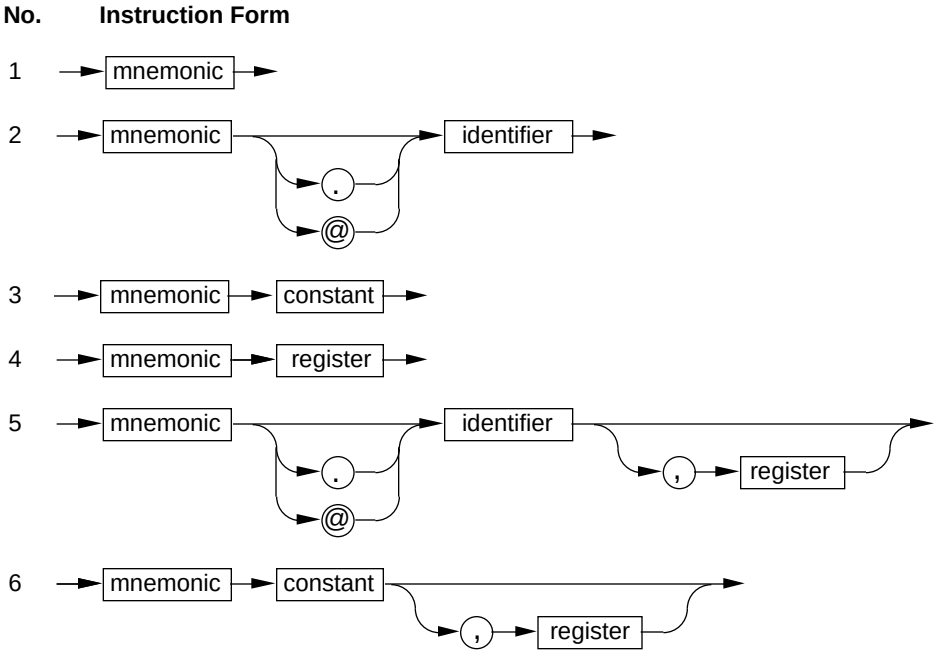


CODE Statement The CODE statement specifies machine-level instructions and pseudocodes to compile into the object file.



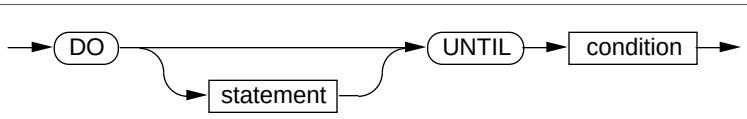
044

instruction



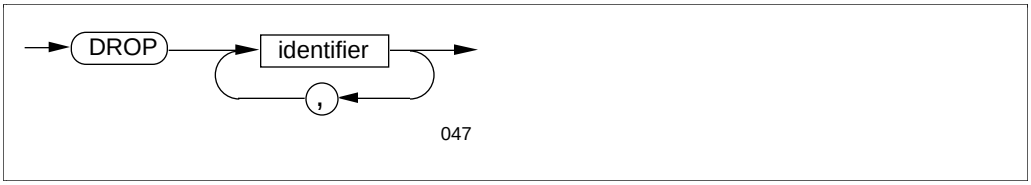
045

DO Statement The DO statement is a posttest loop that repeatedly executes a statement until a specified condition becomes true.

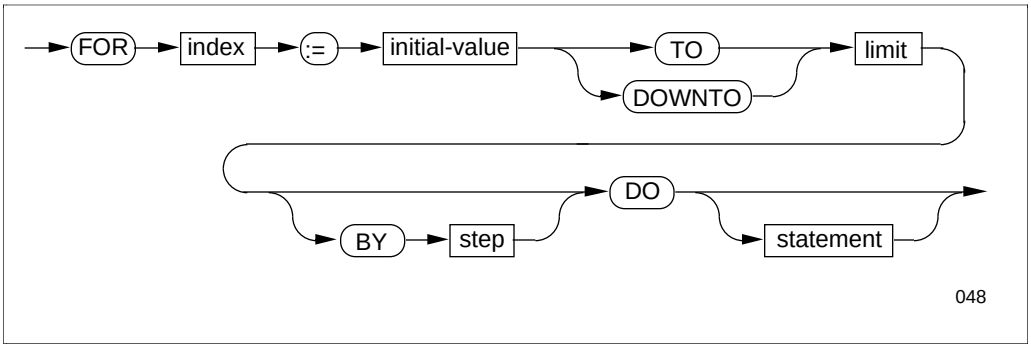


046

DROP Statement The DROP statement disassociates an identifier from an index register reserved by a USE statement or from a label.

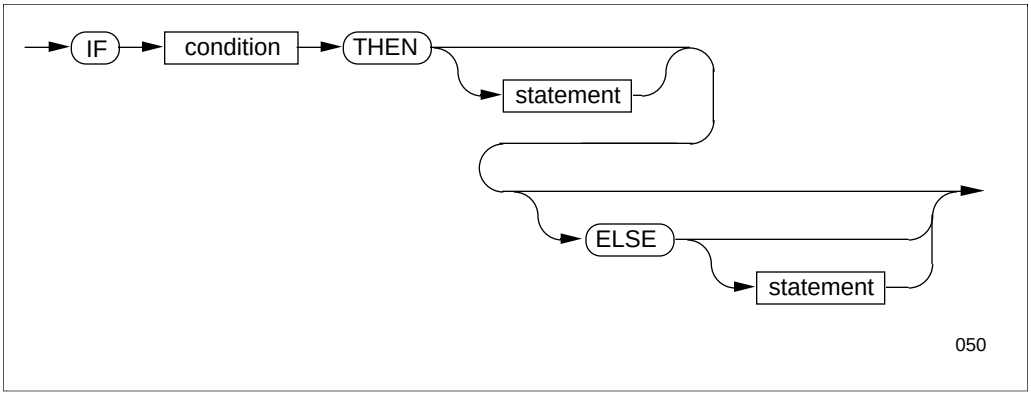


FOR Statement The FOR statement is a pretest loop that repeatedly executes a statement while incrementing or decementing an index.

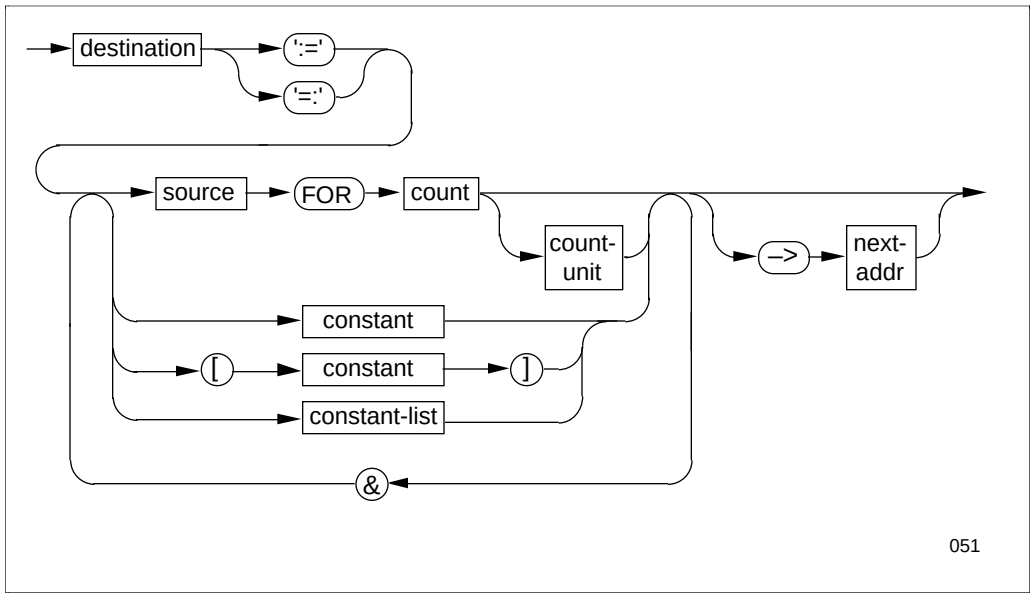


GOTO Statement The GOTO statement unconditionally transfers program control to a labeled statement.

IF Statement The IF statement conditionally selects one of two statements.



Move Statement The move statement copies contiguous bytes, words, or elements to a new location.



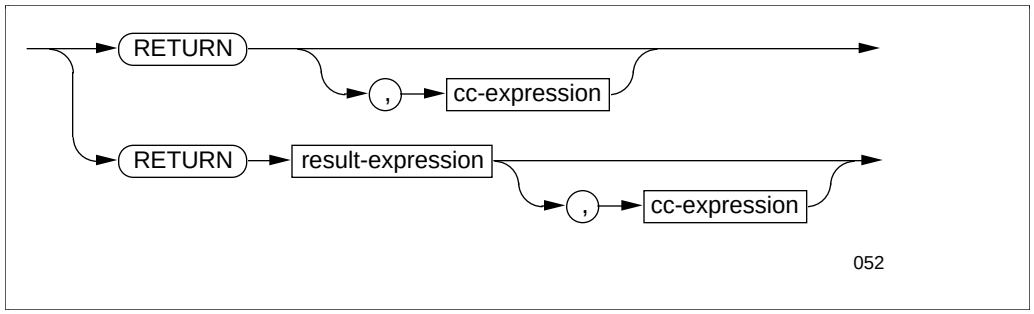
count-unit

BYTES Copies *count* bytes. If *source* and *destination* both have word addresses, BYTES generates a word copy for (*count* + 1) / 2 words.

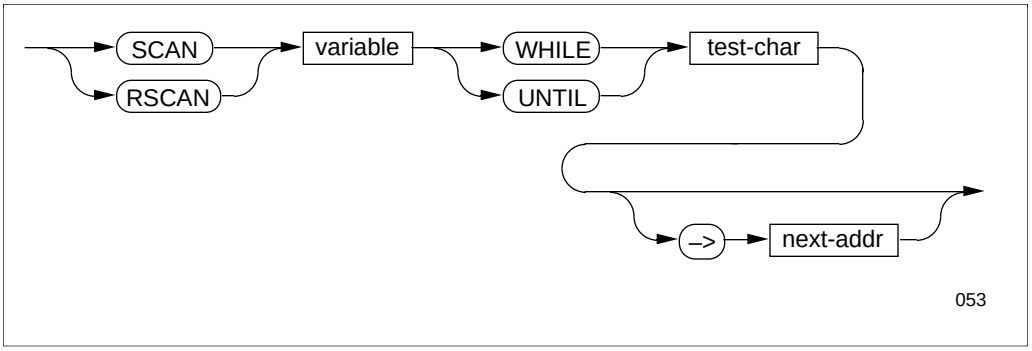
WORDS Copies *count* words.

ELEMENTS Copies *count* occurrences of structures. structure pointers, and substructures. Otherwise, copies *count* units depending on data type.

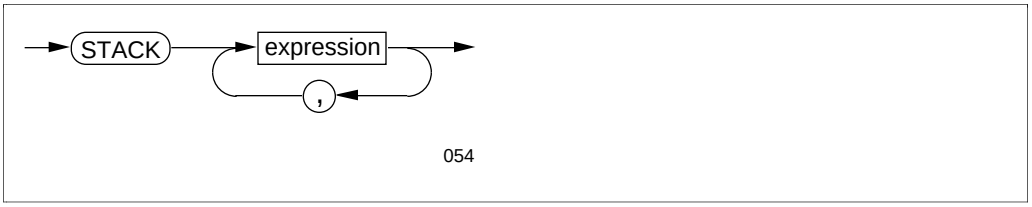
RETURN Statement The RETURN statement returns control to the caller. For a function, RETURN must return a result expression. As of the D20 release, RETURN can also return a condition-code value.



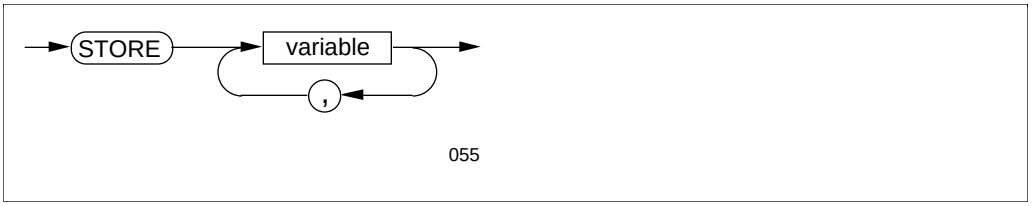
Scan Statement The SCAN or RSCAN statement scans sequential bytes for a test character from left to right or from right to left, respectively.



STACK Statement The STACK statement loads values onto the register stack.



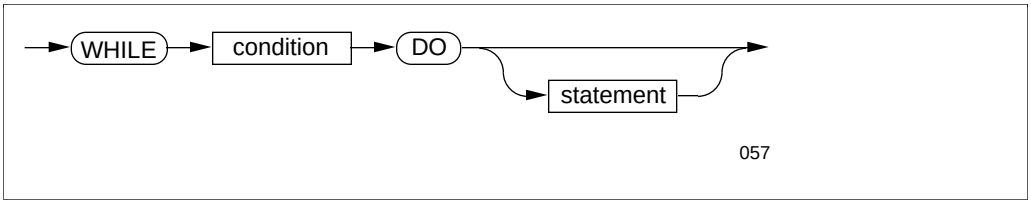
STORE Statement The STORE statement removes values from the register stack and stores them into variables.



USE Statement The USE statement reserves an index register for your use.

WHILE Statement

The WHILE statement is a pretest loop that repeatedly executes a statement while a condition is true.

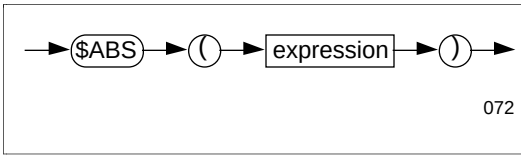


Standard Functions

The following syntax diagrams describe standard functions in alphabetic order.

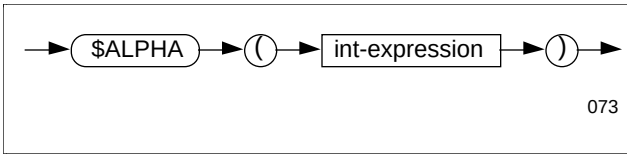
$ABS Function

The $ABS function returns the absolute value of an expression. The returned value has the same data type as the expression.



$ALPHA Function

The $ALPHA function tests the right byte of an INT value for the presence of an alphabetic character.

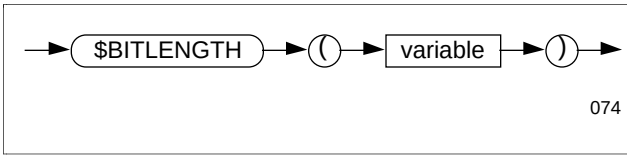


$AXADR Function

See "Privileged Procedures"

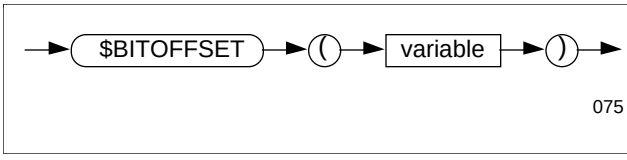
$BITLENGTH Function

The $BITLENGTH function returns the length, in bits, of a variable.



$BITOFFSET Function

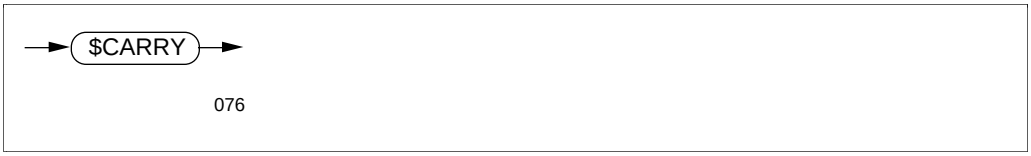
The $BITOFFSET function returns the number of bits from the address of the zeroth structure occurrence to a structure data item.



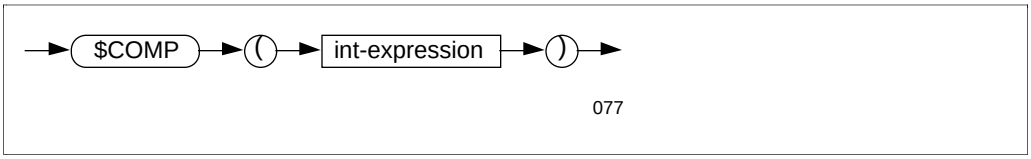
$BOUNDS Function

See "Privileged Procedures"

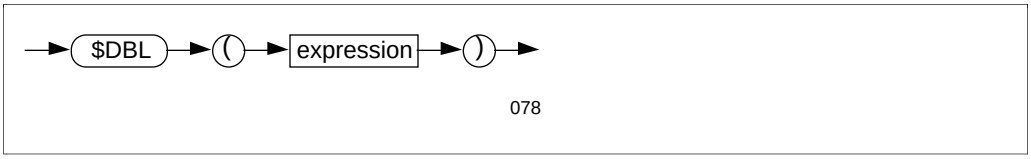
$CARRY Function The $CARRY function checks the state of the carry bit in the environment register and indicates whether a carry out of the high-order bit position occurred.



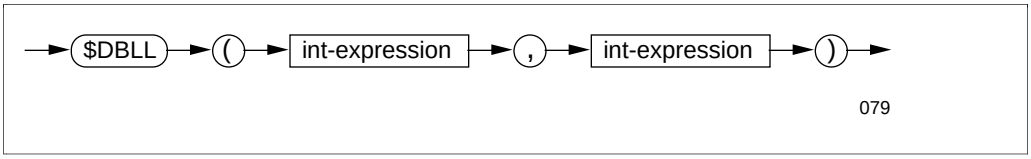
$COMP Function The $COMP function obtains the one's complement of an INT expression.



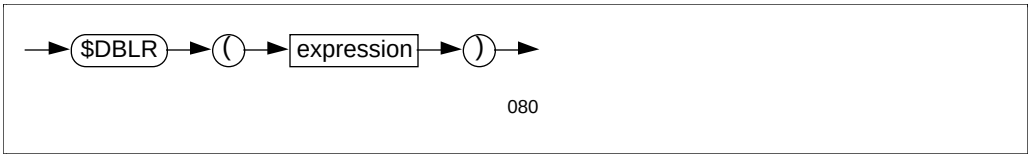
$DBL Function The $DBL function returns an INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression.



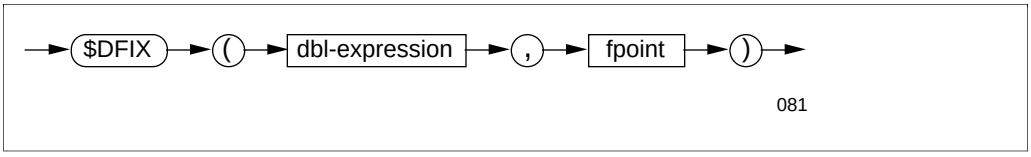
$DBLL Function The $DBLL function returns an INT(32) value from two INT values.



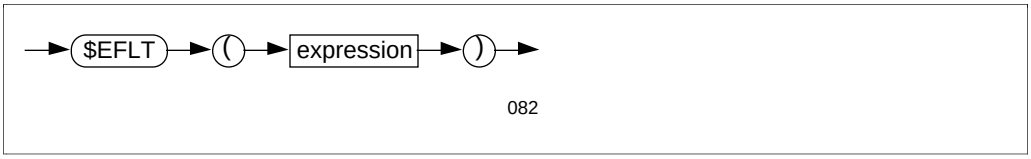
$DBLR Function The $DBLR function returns an INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression and applies rounding to the result.



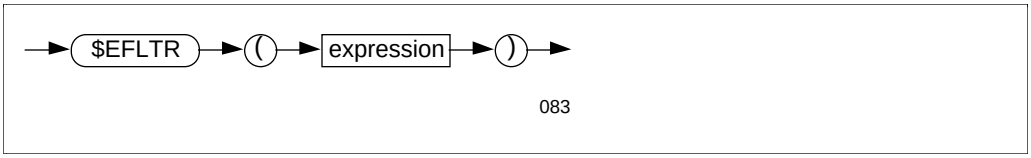
$DFIX Function The $DFIX function returns a FIXED(*fpoint*) expression from an INT(32) expression.



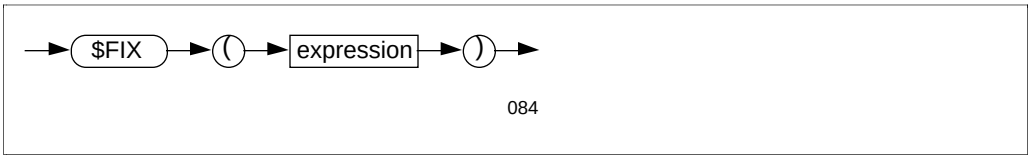
$EFLT Function The $EFLT function returns a REAL(64) value from an INT, INT(32), FIXED(*fpoint*), or REAL expression.



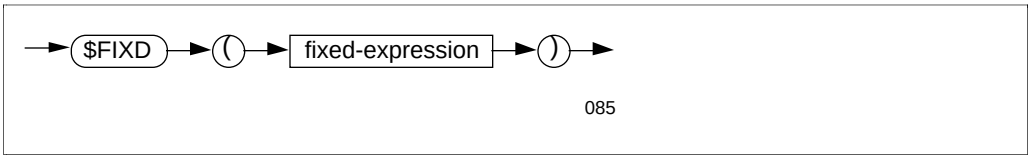
$EFLTR Function The $EFLTR function returns a REAL(64) value from an INT, INT(32), FIXED(*fpoint*), or REAL expression and applies rounding to the result.



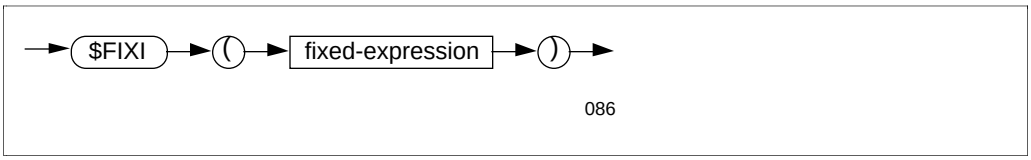
$FIX Function The $FIX function returns a FIXED(0) value from an INT, INT(32), REAL, or REAL(64) expression.



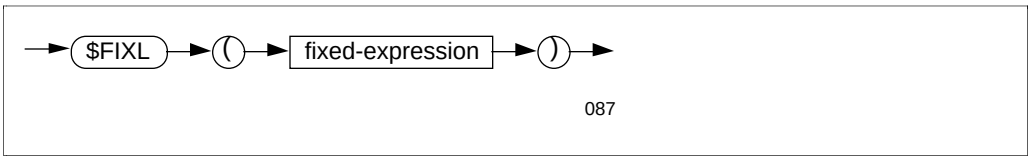
$FIXD Function The $FIXD function returns an INT(32) value from a FIXED(0) expression.



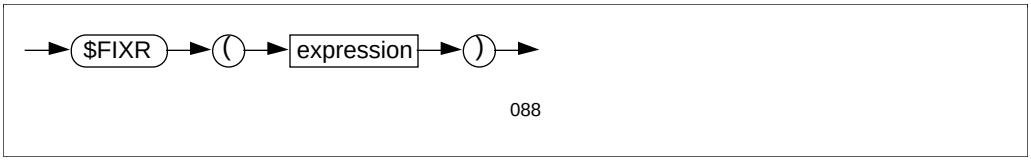
$FIXI Function The $FIXI function returns the signed INT equivalent of a FIXED(0) expression.



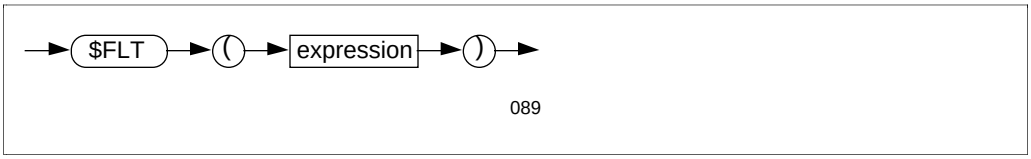
$FIXL Function The $FIXL function returns the unsigned INT equivalent of a FIXED(0) expression.



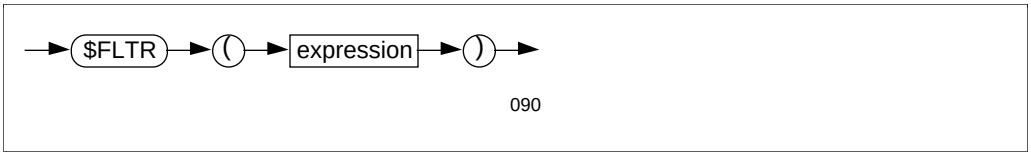
$FIXR Function The $FIXR function returns a FIXED(0) value from an INT, INT(32), FIXED, REAL, or REAL(64) expression and applies rounding to the result.



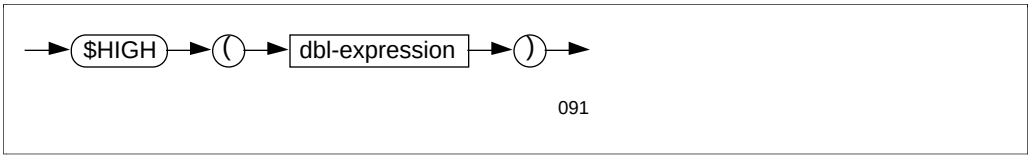
$FLT Function The $FLT function returns a REAL value from an INT, INT(32), FIXED(*fpoint*), or REAL(64) expression.



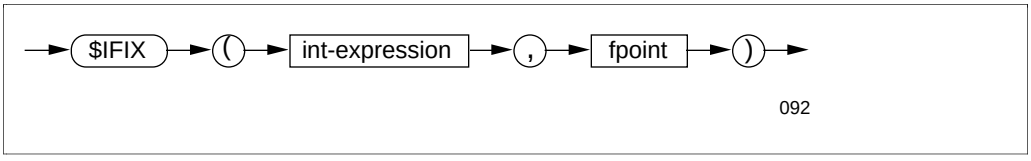
$FLTR Function The $FLTR function returns a REAL value from an INT, INT(32), FIXED(*fpoint*), or REAL(64) expression and applies rounding to the result.



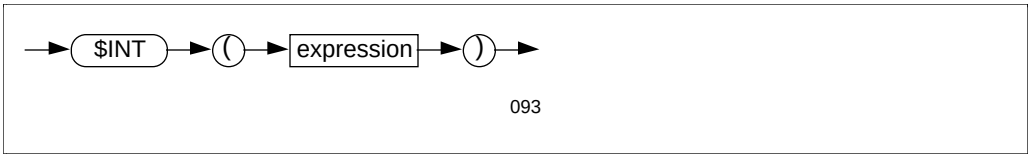
$HIGH Function The $HIGH function returns an INT value that is the high-order 16 bits of an INT(32) expression.



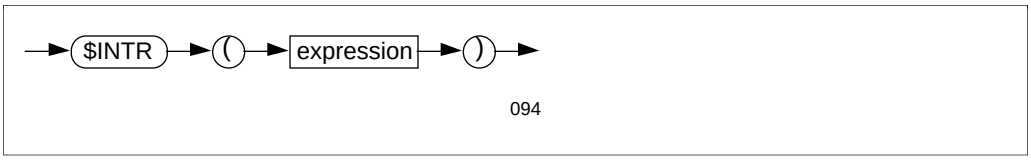
$IFIX Function The $IFIX function returns a FIXED(*fpoint*) value from a signed INT expression.



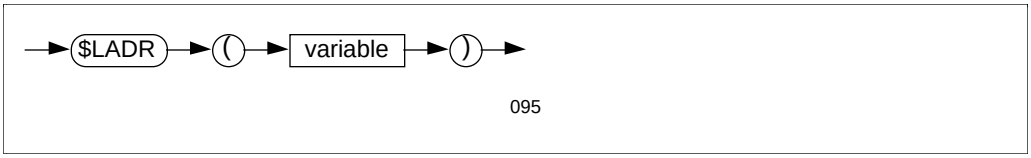
$INT Function The $INT function returns an INT value from the low-order 16 bits of an INT(32 or FIXED(0) expression. $INT returns a fully converted INT expression from a REAL or REAL(64) expression.



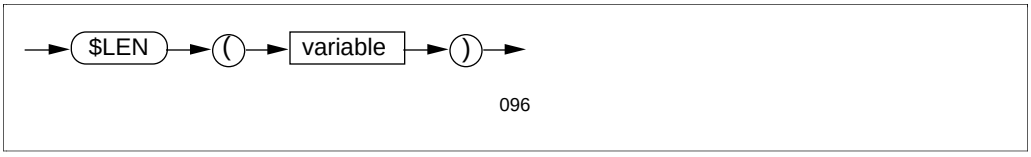
$INTR Function The $INTR function returns an INT value from the low-order 16 bits of an INT(32) or FIXED(0) expression. $INTR returns a fully converted and rounded INT expression from a REAL or REAL(64) expression.



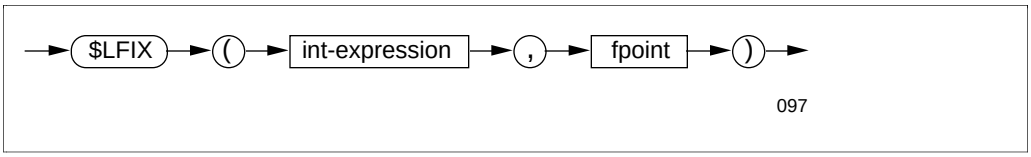
$LADR Function The $LADR function returns the standard (16-bit) address of a variable that is accessed through an extended (32-bit) pointer.



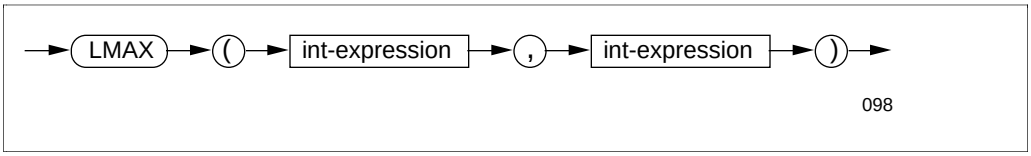
$LEN Function The $LEN function returns the length, in bytes, of one occurrence of a variable.



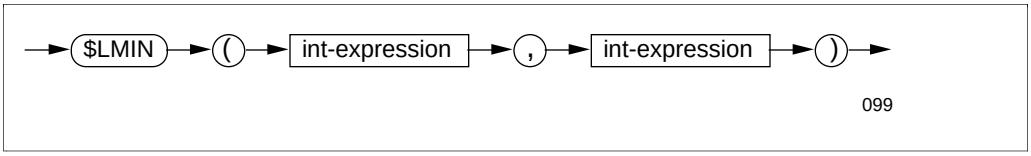
$LFIX Function The $LFIX function returns a FIXED(*fpoint*) expression from an unsigned INT expression.



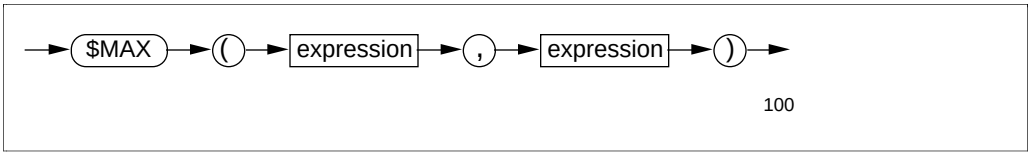
$LMAX Function The $LMAX function returns the maximum of two unsigned INT expressions.



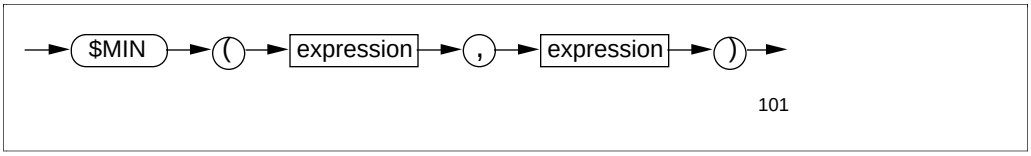
$LMIN Function The $LMIN function returns the minimum of two unsigned INT expressions.



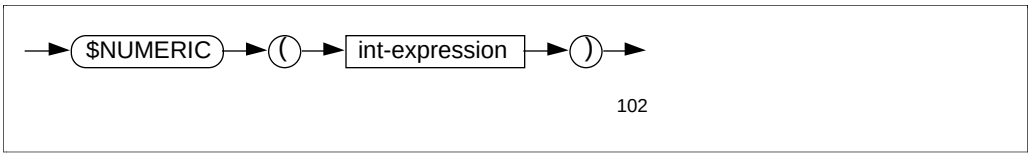
$MAX Function The $MAX function returns the maximum of two signed INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expressions.



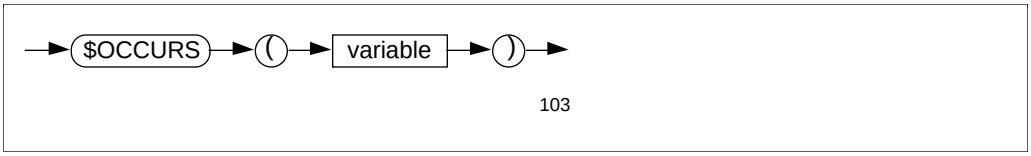
$MIN Function The $MIN function returns the minimum of two INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expressions.



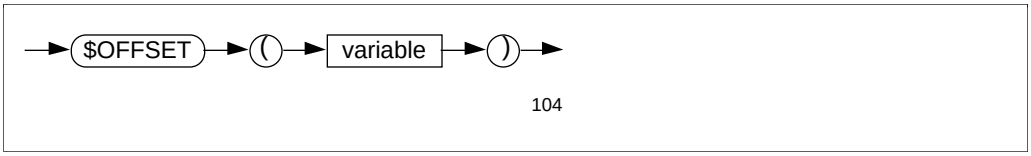
$NUMERIC Function The $NUMERIC function tests the right half of an INT value for the presence of an ASCII numeric character.



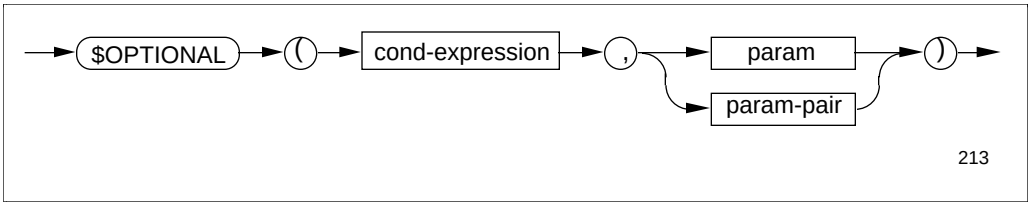
$OCCURS Function The $OCCURS function returns the number of occurrences of a variable.



$OFFSET Function The $OFFSET function returns the number of bytes from the address of the zeroth structure occurrence to a structure data item.

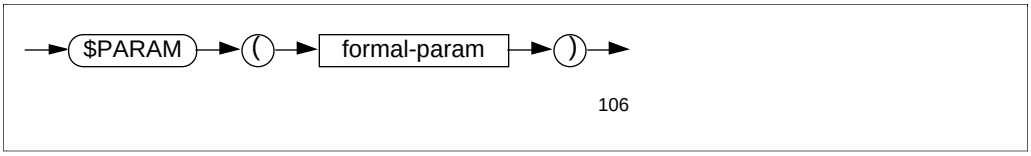


$OPTIONAL Function The $OPTIONAL function controls whether a given parameter or parameter pair is passed to a VARIABLE or EXTENSIBLE procedure. OPTIONAL is a D20 or later feature.

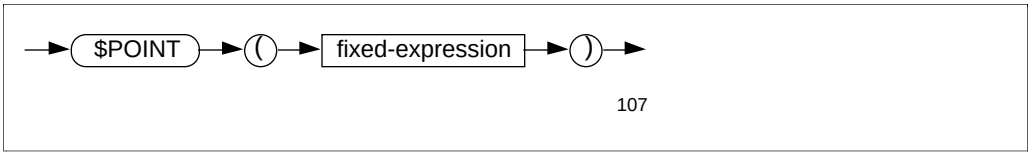


$OVERFLOW Function The $OVERFLOW function checks the state of the overflow indicator and indicates whether an overflow occurred during an arithmetic operation.

$PARAM Function The $PARAM function checks for the presence or absence of an actual parameter in the call that invoked the current procedure or subprocedure.

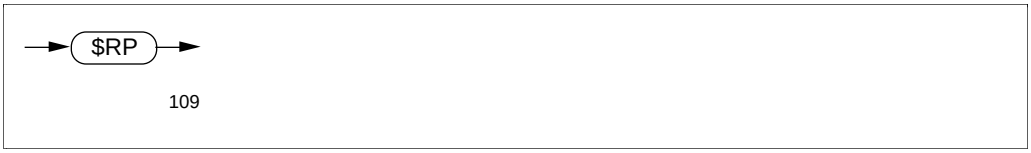


$POINT Function The $POINT function returns the *fpoint* value, in integer form, associated with a FIXED expression.

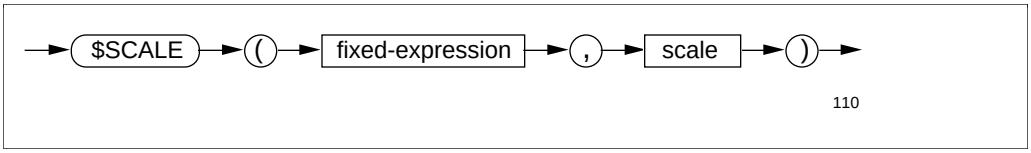


$READCLOCK Function The $READCLOCK function returns the current setting of the system clock.

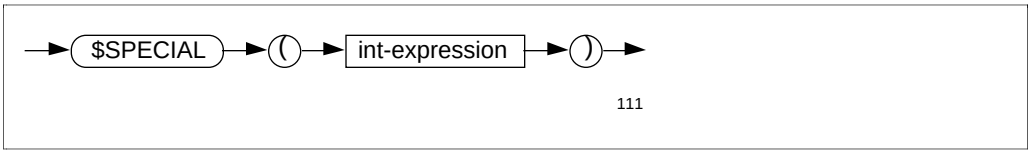
$RP Function The $RP function returns the current setting of the compiler's internal RP counter. (RP is the register stack pointer.)



$SCALE Function The $SCALE function moves the position of the implied decimal point by adjusting the internal representation of a FIXED(*fpoint*) expression.

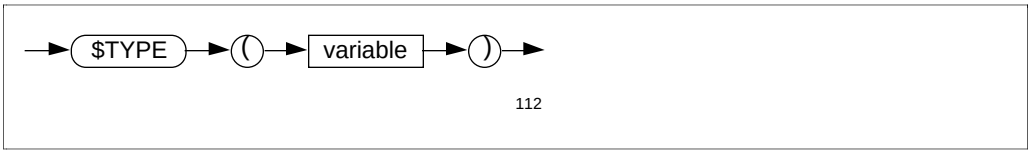


$SPECIAL Function The $SPECIAL function tests the right half of an INT value for the presence of an ASCII special (non-alphanumeric) character.

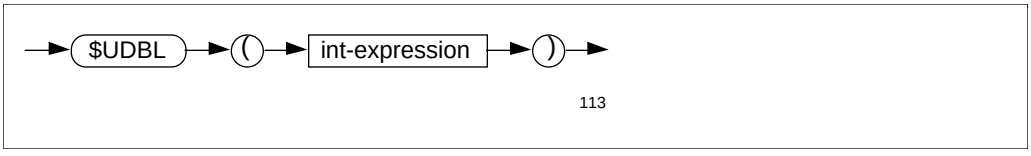


$SWITCHES Function See "Privileged Procedures"

$TYPE Function The $TYPE function returns a value that indicates the data type of a variable.

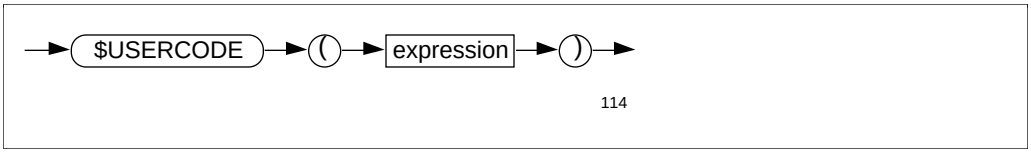


$UDBL Function The $UDBL function returns an INT(32) value from an unsigned INT expression.



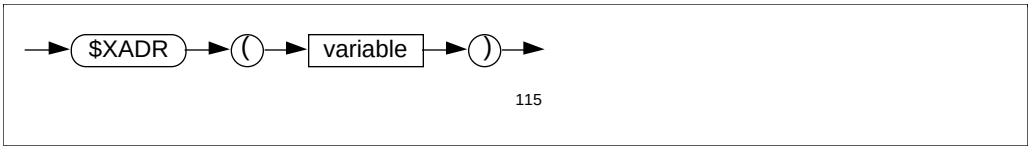
$USERCODE Function

The $USERCODE function returns the content of the word at the specified location in the current user code segment.



$XADR Function

The $XADR function converts a standard address to an extended address.

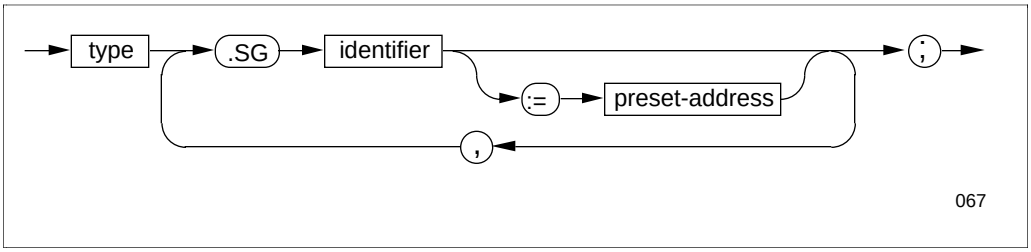


Privileged Procedures

The following syntax diagrams describe declarations for privileged procedures.

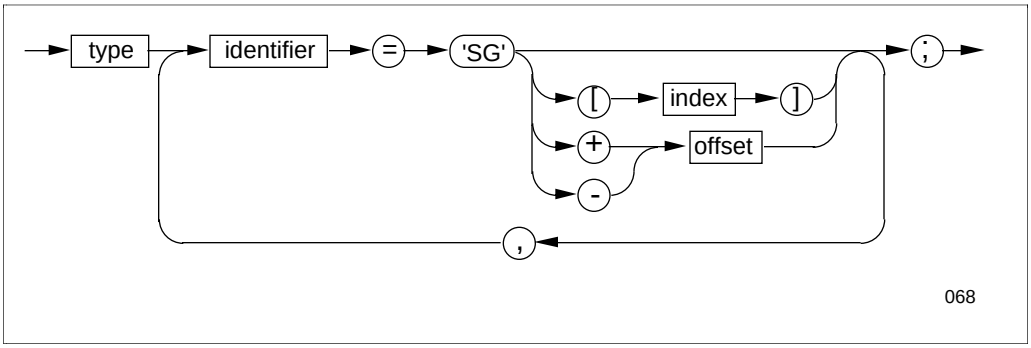
System Global Pointers

The system global pointer declaration associates an identifier with a memory location that you load with the address of a variable located in the system global data area.



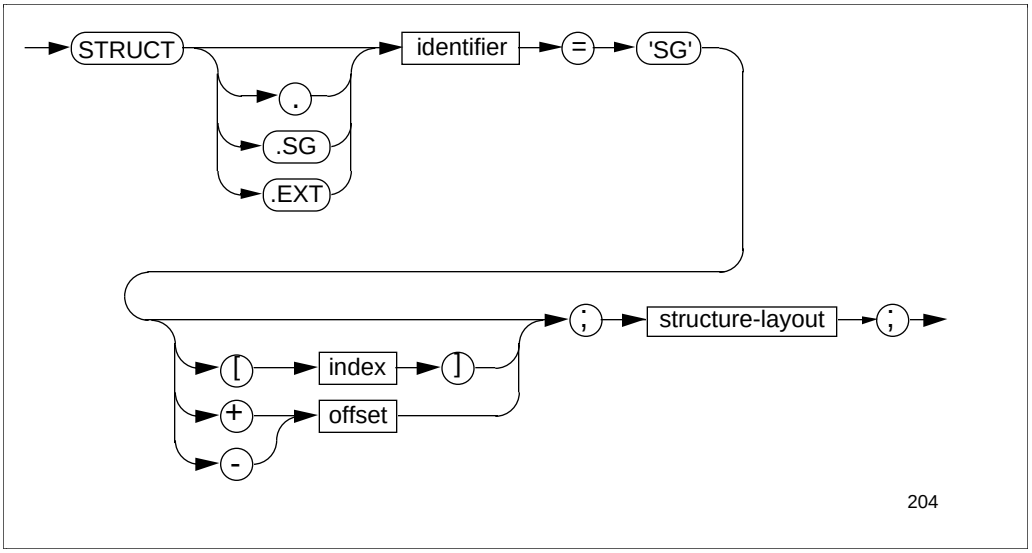
'SG'-Equivalenced Simple Variables

The 'SG'-equivalenced simple variable declaration associates a simple variable with a location that is relative to the base address of the system global data area.



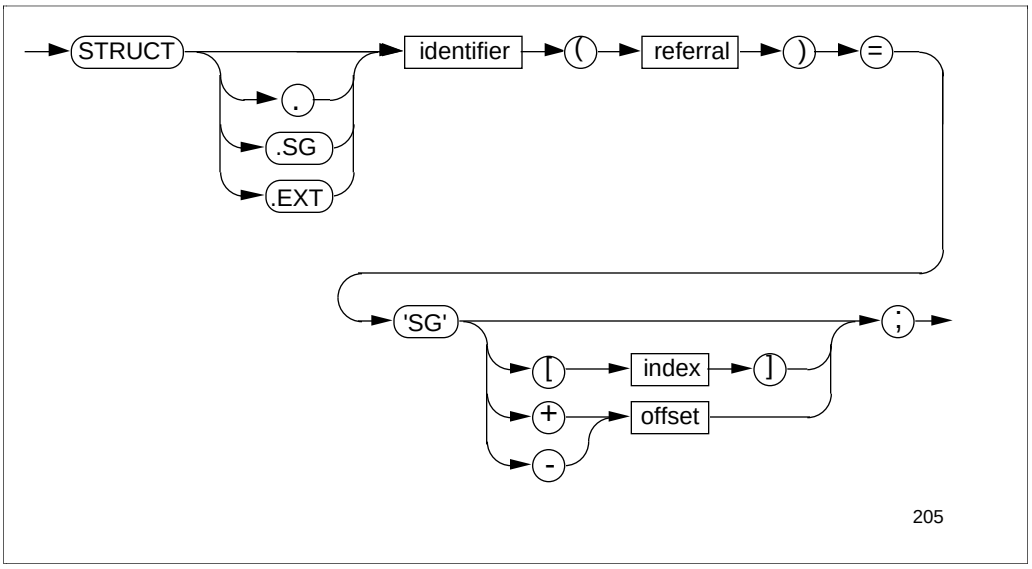
'SG'-Equivalenced Definition Structures

The 'SG'-equivalenced definition structure declaration associates a definition structure with a location relative to the base address of the system global data area.



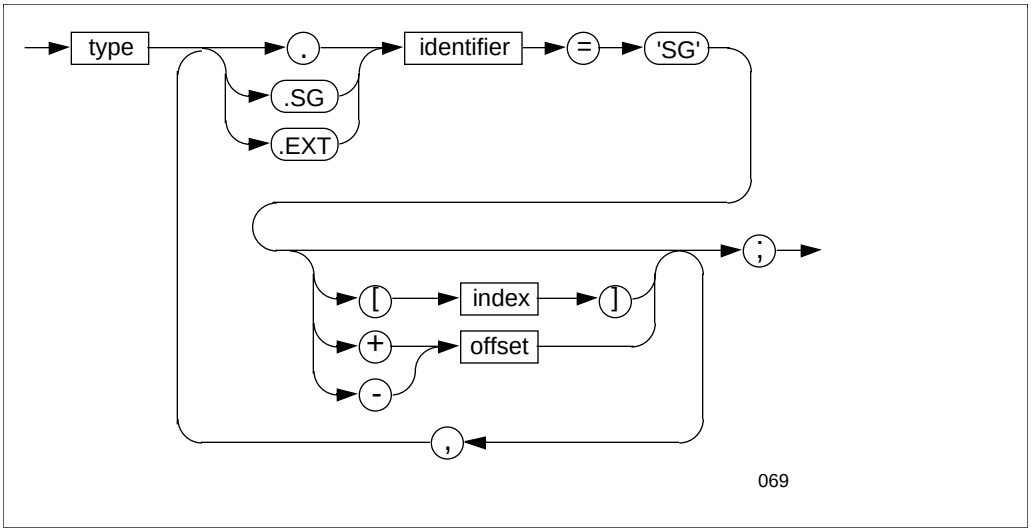
'SG'-Equivalenced Referral Structures

The 'SG'-equivalenced referral structure declaration associates a referral structure with a location relative to the base address of the system global data area.



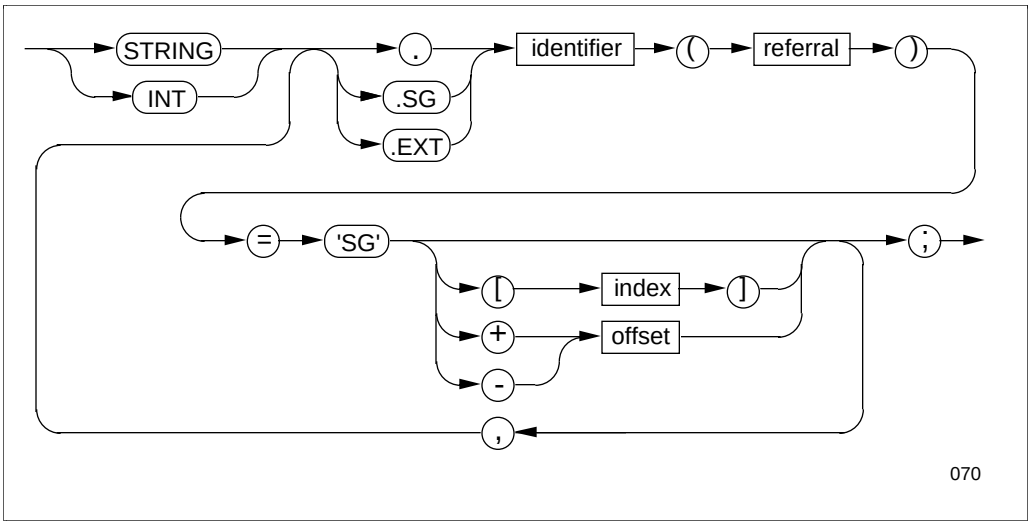
'SG'-Equivalenced Simple Pointers

The 'SG'-equivalenced simple pointer declaration associates a simple pointer with a location relative to the base address of the system global data area.



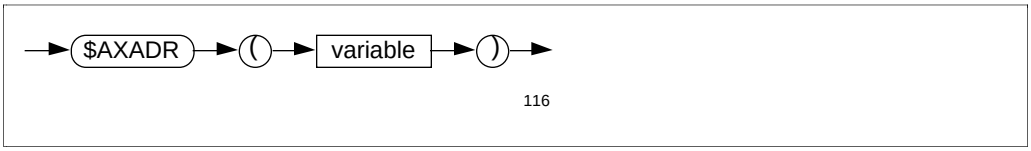
'SG'-Equivalenced Structure Pointers

The 'SG'-equivalenced structure pointer declaration associates a structure pointer with a location relative to the base address of the system global data area.

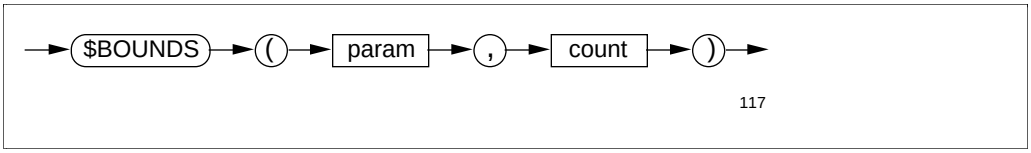


$AXADR Function

The $AXADR function returns an absolute extended address.

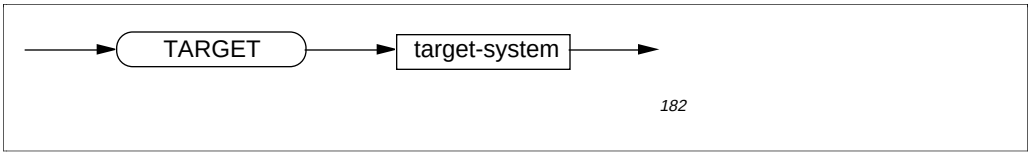


$BOUNDS Function The $BOUNDS function checks the location of a parameter passed to a system procedure to prevent a pointer that contains an incorrect address from overlaying the stack (S) register with data.



$SWITCHES Function The $SWITCHES function returns the current content of the switch register.

TARGET Directive The TARGET directive specifies the target system for which you have written conditional code. TARGET works in conjunction with the IF and ENDIF directives in D20 or later object files.



target-system

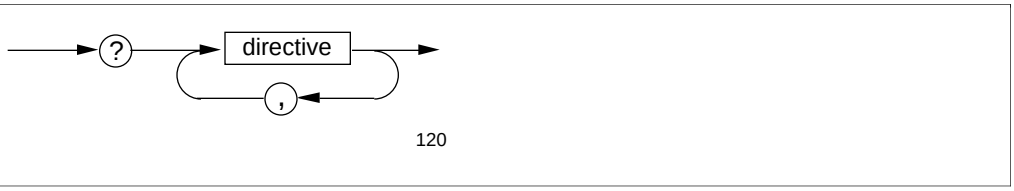
ANY	Compile IF ANY conditional code; specifies that the conditional code as written is not system-dependent.
TNS_ARCH	Compile IF TNS_ARCH conditional code.
TNS_R_ARCH	Compile IF TNS_R_ARCH conditional code.

Compiler Directives

The following syntax diagrams describe directive lines, followed by compiler directives in alphabetic order.

Directive Lines

A directive line in your source code contains one or more compiler directives.



ABORT Directive

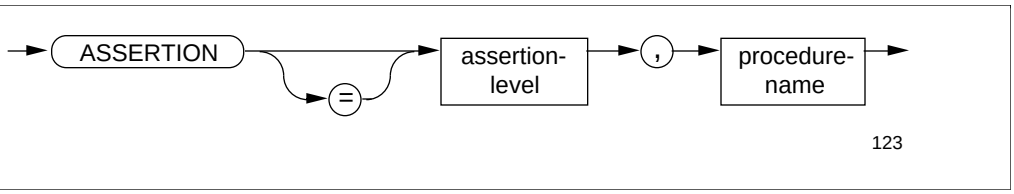
The ABORT directive terminates compilation if the compiler cannot open a file specified in a SOURCE directive. The default is ABORT.

ABSLIST Directive

ABSLIST lists code addresses relative to the code area base. The default is NOABSLIST.

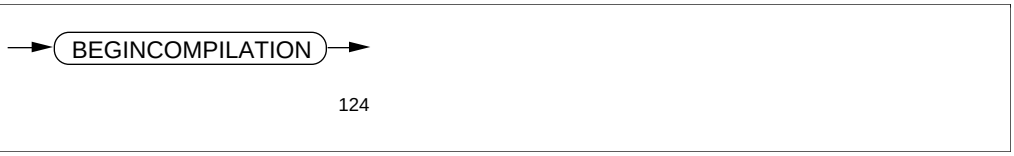
ASSERTION Directive

ASSERTION invokes a procedure when a condition defined in an ASSERT statement is true.

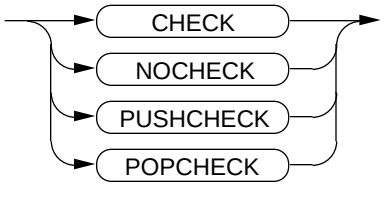


BEGINCOMPILATION Directive

BEGINCOMPILATION marks the point in the source file where compilation is to begin if the USEGLOBALS directive is in effect.

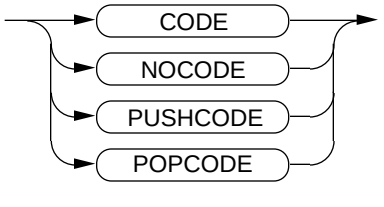


CHECK Directive CHECK generates range-checking code for certain features. The default is NOCHECK.



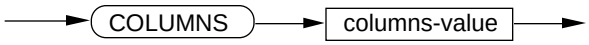
125

CODE Directive CODE lists instruction codes and constants in octal format after each procedure. The default is CODE.



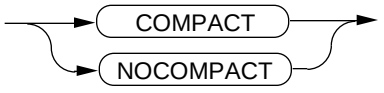
126

COLUMNS Directive COLUMNS directs the compiler to treat any text beyond the specified column as comments.



127

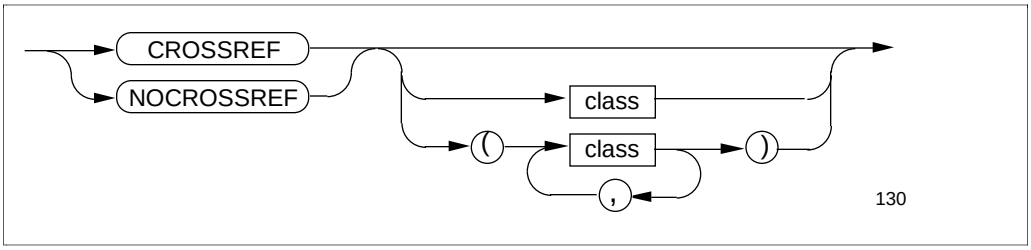
COMPACT Directive COMPACT moves procedures into gaps below the 32K-word boundary of the code area if they fit. The default is COMPACT.



128

CPU Directive CPU specifies that the object file runs on a TNS system. (The need for this directive no longer exists. This directive has no effect on the object file and is retained only for compatibility with programs that still specify it.)

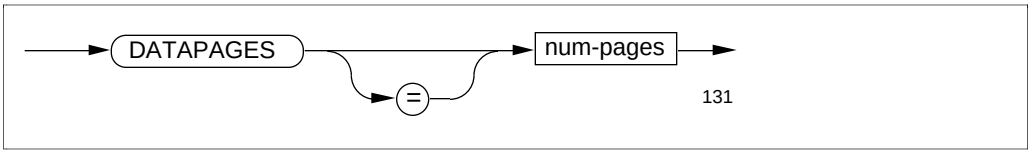
CROSSREF Directive CROSSREF collects source-level declarations and cross-reference information or specifies CROSSREF classes. The default is NOCROSSREF.



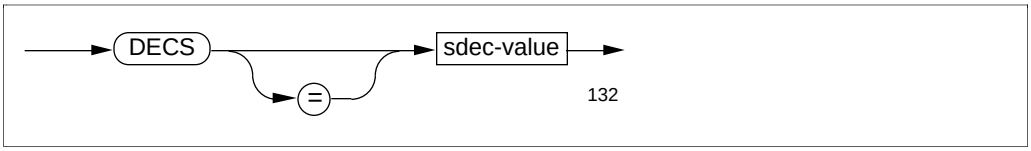
class

BLOCKS	Named and private data blocks
DEFINES	Named text
LABELS	Statement labels
LITERALS	Named constants
PROCEDURES	Procedures
PROCPARAMS	Procedures that are formal parameters
SUBPROCS	Subprocedures
TEMPLATES	Template structures
UNREF	Unreferenced identifiers
VARIABLES	Simple variables, arrays, definition structures, referral structures, pointers, and equivalenced variables

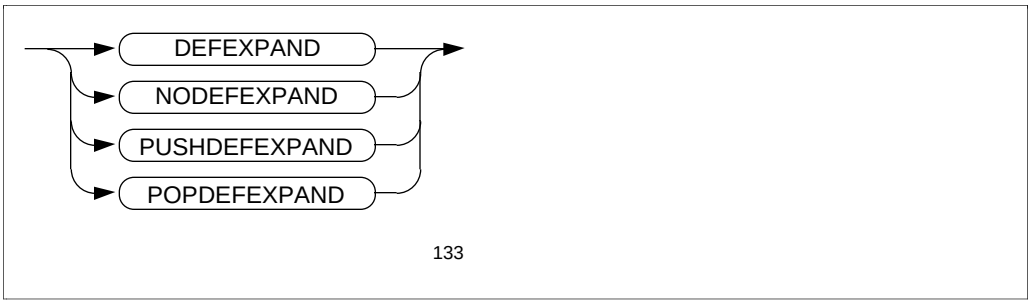
DATAPAGES Directive DATAPAGES sets the size of the data area in the user data segment.



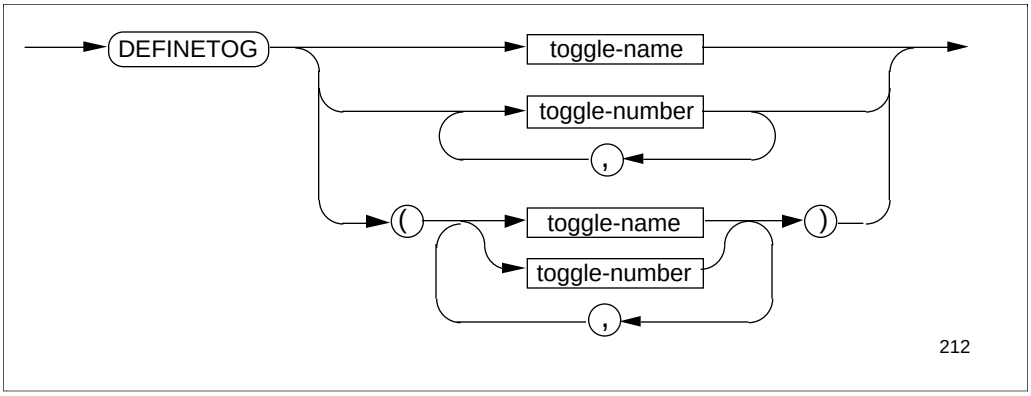
DECS Directive DECS decrements the compiler's internal S-register counter.



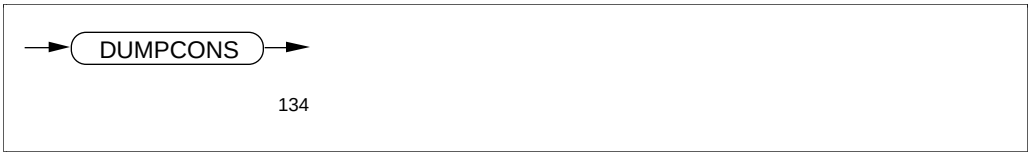
DEFEXPAND Directive DEFEXPAND lists expanded DEFINEs and SQL-TAL code in the compiler listing. The default is NODEFEXPAND.



DEFINETOG Directive DEFINETOG specifies named or numeric toggles, without changing any prior settings, for use in conditional compilation. DEFINETOG is a D20 or later feature.

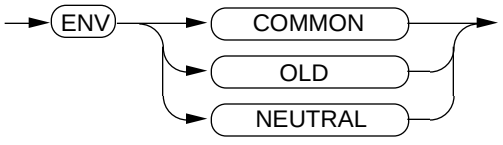


DUMPCONS Directive DUMPCONS inserts the contents of the compiler's constant table into the object code.



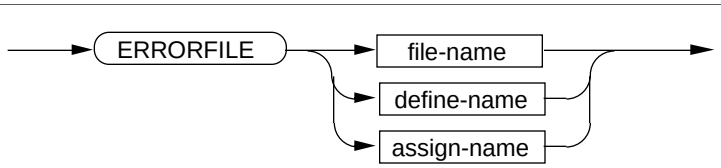
ENDIF Directive See "IF and ENDIF Directives."

ENV Directive ENV specifies the intended run-time environment of a D-series object file. The default is ENV NEUTRAL.

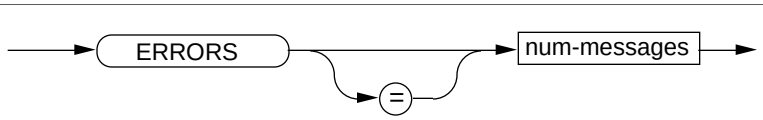


Attribute	Intended Run-Time Environment
COMMON	The CRE
OLD	A COBOL or FORTRAN run-time environment outside the CRE
NEUTRAL	None; the program relies primarily on system procedures

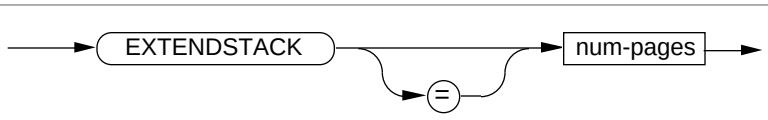
ERRORFILE Directive ERRORFILE logs compilation errors and warnings to an error file so you can use the TACL FIXERRS macro to view the diagnostic messages in one PS Text Edit window and correct the source file in another window.



ERRORS Directive ERRORS sets the maximum number of error messages to allow before the compiler terminates the compilation.

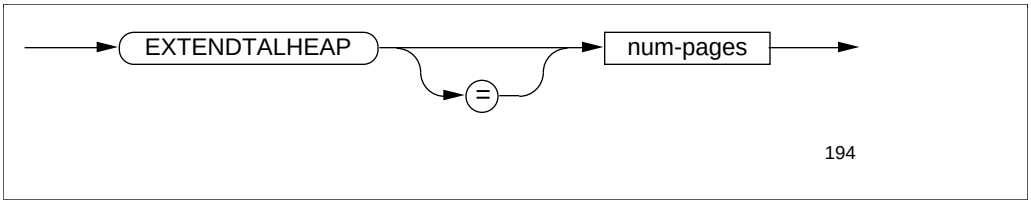


EXTENDSTACK Directive EXTENDSTACK increases the size of the data stack in the user data segment.



EXTENDTALHEAP Directive

EXTENDTALHEAP increases the size of the compiler's internal heap for a D-series compilation unit.



FIXUP Directive

FIXUP directs BINSERV to perform its fixup step. The default is FIXUP.

FMAP Directive

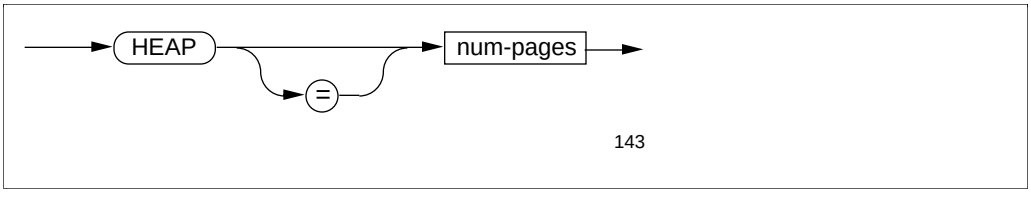
FMAP lists the file map. The default is NOFMAP.

GMAP Directive

GMAP lists the global map. The default is GMAP.

HEAP Directive

HEAP sets the size of the CRE user heap for a D-series compilation unit if the ENV COMMON directive is in effect.



HIGHPIN Directive

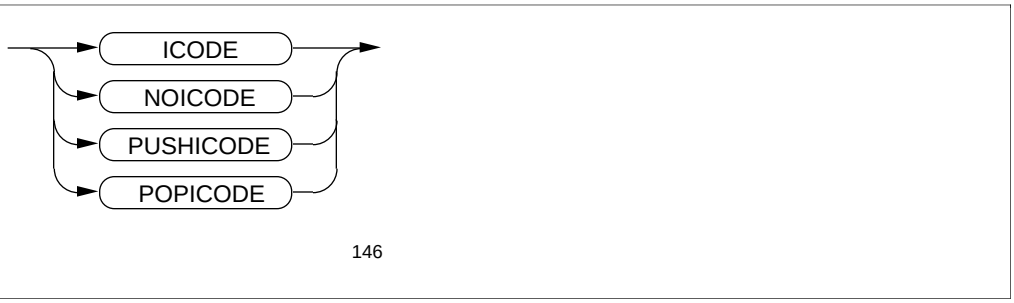
HIGHPIN sets the HIGHPIN attribute in a D-series object file.

HIGHREQUESTERS Directive

HIGHREQUESTERS sets the HIGHREQUESTERS attribute in a D-series object file.

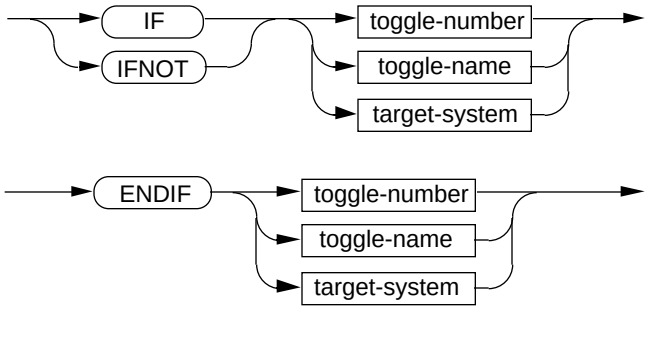
ICODE Directive

ICODE lists the instruction-code (icode) mnemonics for subsequent procedures. The default is NOICODE.



IF and ENDIF Directives

IF and IFNOT control conditional compilation based on a condition. The ENDIF directive terminates the range of the matching IF or IFNOT directive. The D20 or later release supports named toggles and target-system toggles.



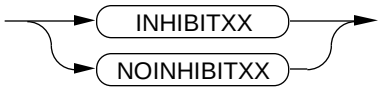
147

target-system

ANY	ANY is specified in a TARGET directive in this compilation.
TARGETSPECIFIED	A TARGET directive appears in this compilation.
TNS_ARCH	TNS_ARCH is specified in a TARGET directive .
TNS_R_ARCH	TNS_R_ARCH is specified in a TARGET directive.

INHIBITXX Directive

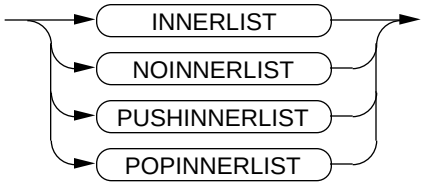
INHIBITXX generates inefficient but correct code for extended global declarations in relocatable blocks that Binder might locate after the first 64 words of the primary global area of the user data segment. The default is NOINHIBITXX.



148

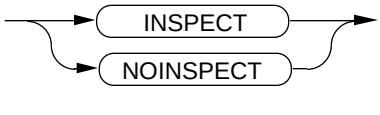
INNERLIST Directive

INNERLIST lists the instruction code mnemonics (and the compiler's RP setting) for each statement. The default is NOINNERLIST.



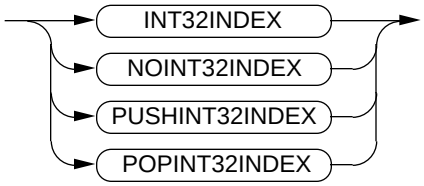
149

INSPECT Directive INSPECT sets the Inspect product as the default debugger for the object file. The default is NOINSPECT.



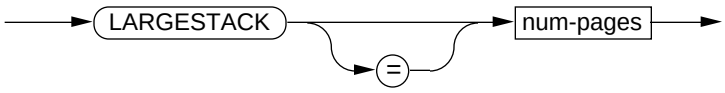
150

INT32INDEX Directive INT32INDEX generates INT(32) indexes from INT indexes for accessing items in an extended indirect structure in a D-series program. The default is NOINT32INDEX.



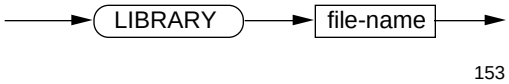
151

LARGESTACK Directive LARGESTACK sets the size of the extended stack in the automatic extended data segment.



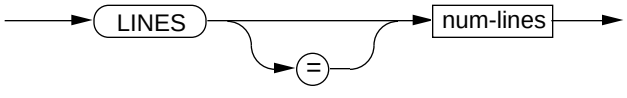
152

LIBRARY Directive LIBRARY specifies the name of the TNS software user run-time library to be associated with the object file at run time.



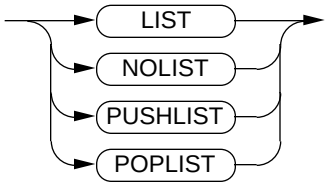
153

LINES Directive LINES sets the maximum number of output lines per page.



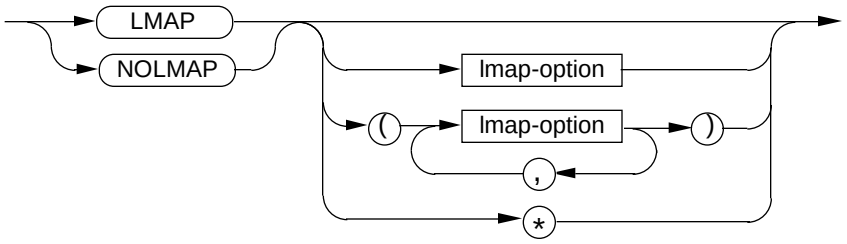
154

LIST Directive LIST lists the source text for subsequent source code if NOSUPPRESS is in effect. The default is LIST.



155

LMAP Directive LMAP lists load-map and cross-reference information. The default is LMAP ALPHA.

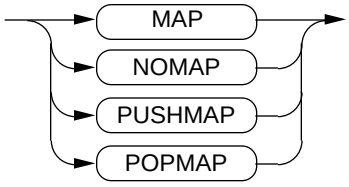


156

lmap-option

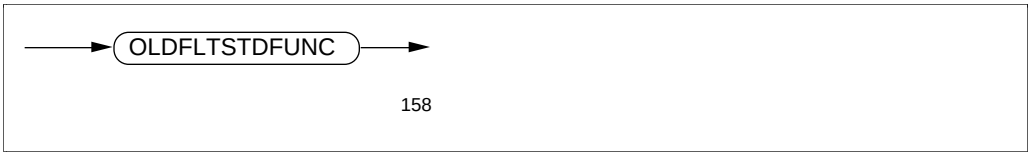
ALPHA	List load maps of procedures and data blocks sorted by name
LOC	List ALPHA maps and load maps of procedures and data blocks sorted by starting address
XREF	List ALPHA maps and entry-point and common data-block cross-references for object file
*	List ALPHA, LOC, and XREF maps.

MAP Directive MAP lists the identifier maps. The default is MAP.

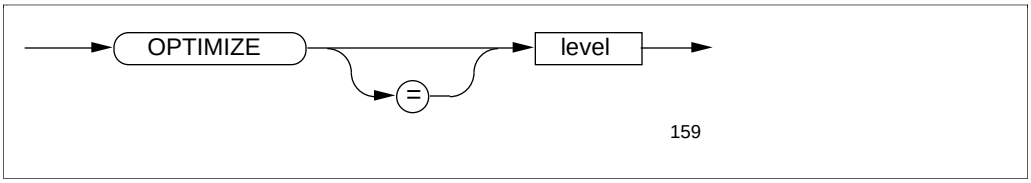


157

OLDFLTSTDFUNC Directive OLDFLTSTDFUNC treats arguments to the $FLT, $FLTR, $EFLT, and $EFLTR standard functions as if they were FIXED(0) values.



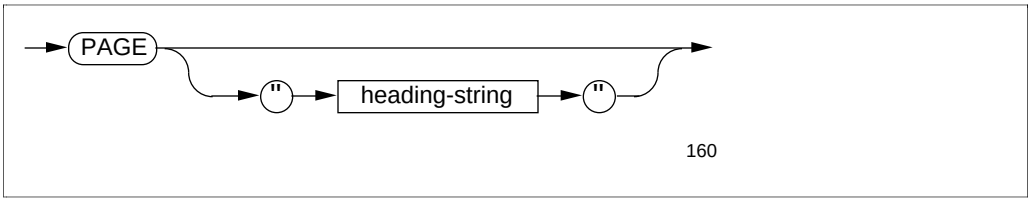
OPTIMIZE Directive OPTIMIZE specifies the level at which the compiler optimizes the object code.



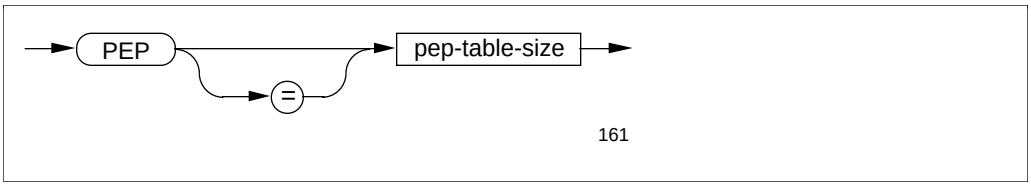
level

- 0 None
- 1 Within a statement
- 2 Within and across statement boundaries

PAGE Directive PAGE optionally prints a heading and causes a page eject.



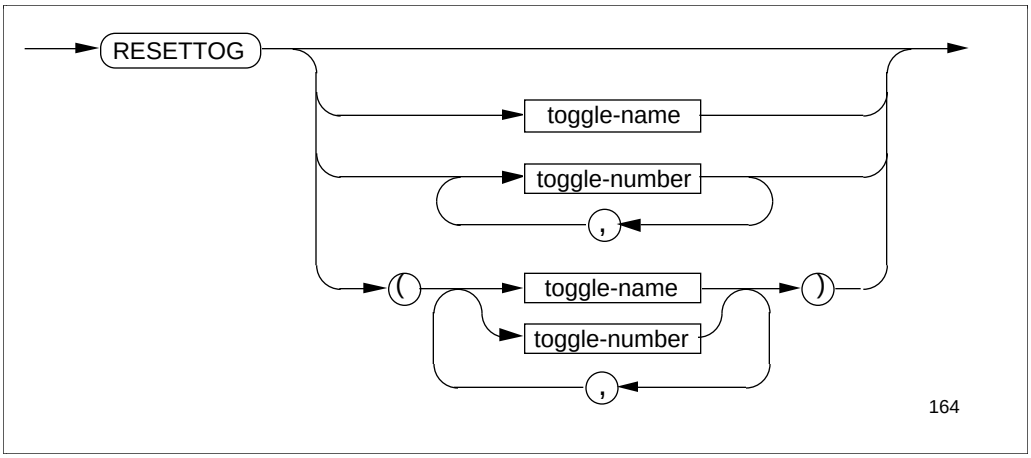
PEP Directive PEP specifies the size of the procedure entry-point (PEP) table.



PRINTSYM Directive PRINTSYM lists symbols. The default is PRINTSYM.

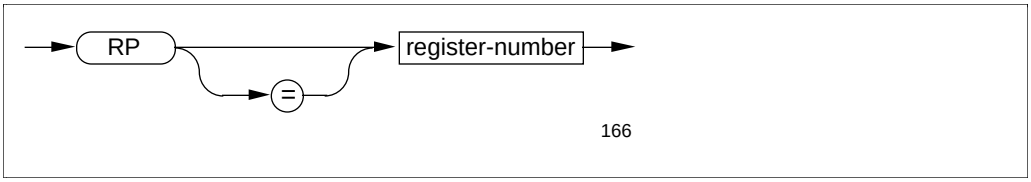
RELOCATE Directive RELOCATE lists BINSERV warnings for declarations that depend on absolute addresses in the primary global data area of the user data segment.

RESETTOG Directive RESETTOG creates new toggles in the off state and turns off toggles created by SETTOG. As of the D20 release, RESETTOG supports named toggles.



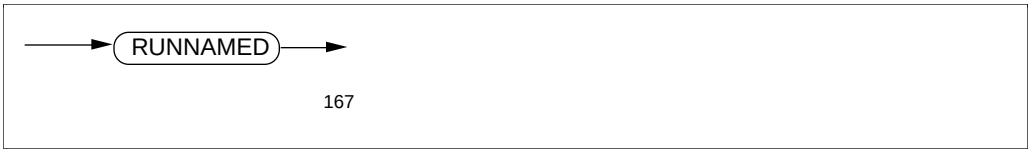
ROUND Directive ROUND rounds FIXED values assigned to FIXED variables that have smaller *fpoint* values than the values you are assigning. The default is NOROUND.

RP Directive RP sets the compiler's internal register pointer (RP) count. RP tells the compiler how many registers are currently in use on the register stack.



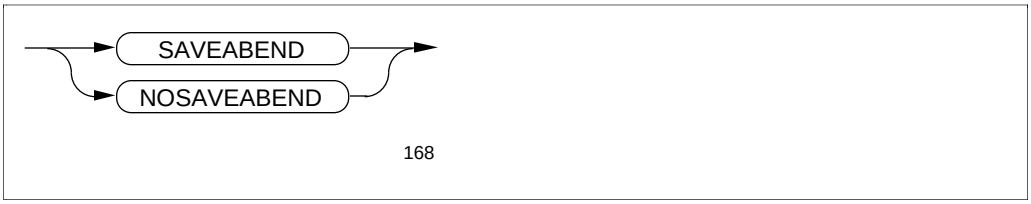
RUNNAMED Directive

RUNNAMED causes a D-series object file to run on a D-series system as a named process even if you do not provide a name for it.



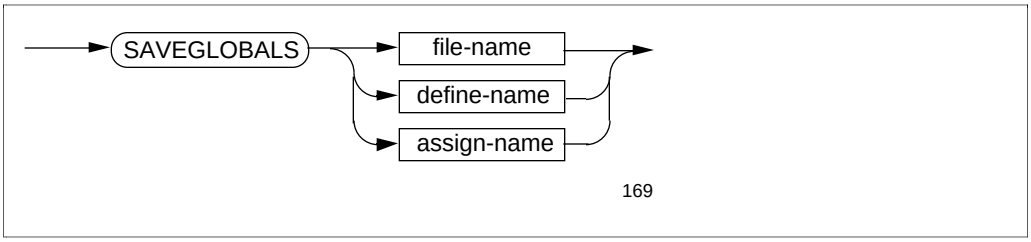
SAVEABEND Directive

SAVEABEND directs the Inspect product to generate a save file if your process terminates abnormally during execution. The default is NOSAVEABEND.



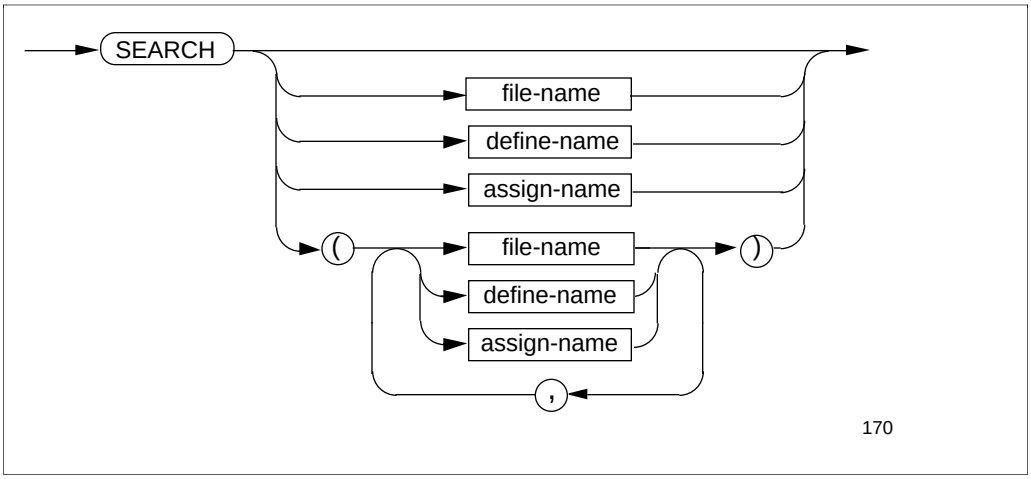
SAVEGLOBALS Directive

SAVEGLOBALS saves all global data declarations in a file for use in subsequent compilations that specify the USEGLOBALS directive.

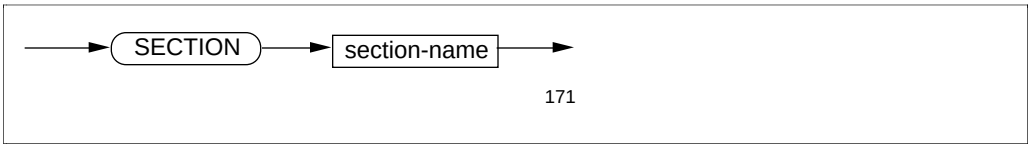


SEARCH Directive

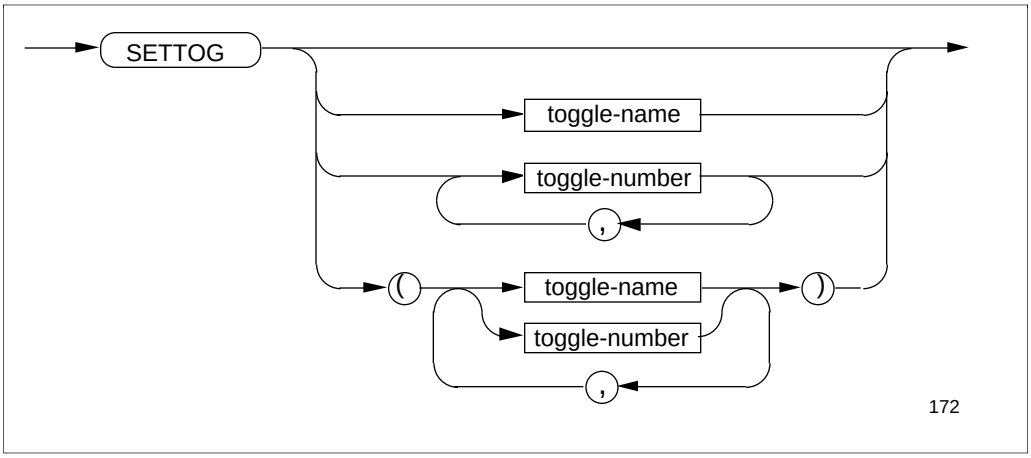
SEARCH specifies object files from which BINSERV can resolve unsatisfied external references and validate parameter lists at the end of compilation. By default, BINSERV does not attempt to resolve unsatisfied external references.



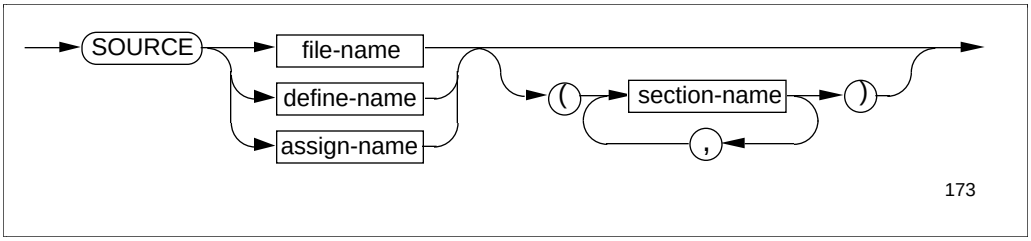
SECTION Directive SECTION gives a name to a section of a source file for use in a SOURCE directive.



SETTOG Directive SETTOG turns the specified toggles on for use in conditional compilations. As of the D20 release, SETTOG supports named toggles.



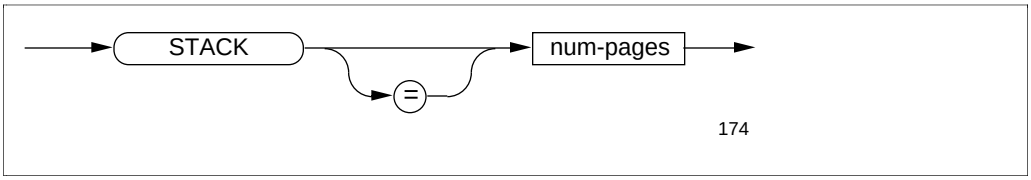
SOURCE Directive SOURCE specifies source code to include from another source file.



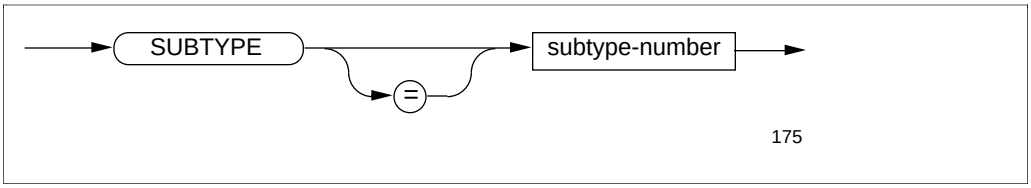
SQL Directive See the *NonStop SQL Programming Manual for TAL.*

SQLMEM Directive See the *NonStop SQL Programming Manual for TAL.*

STACK Directive STACK sets the size of the data stack in the user data segment.

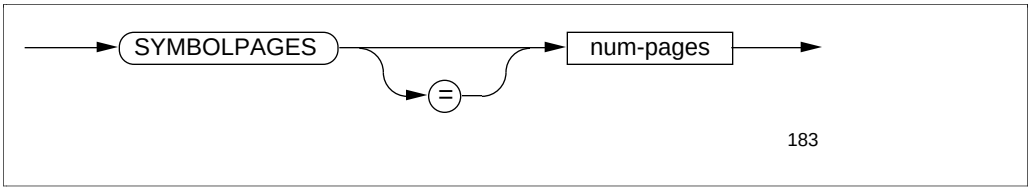


SUBTYPE Directive SUBTYPE specifies that the object file is to execute as a process of a specified subtype.



SUPPRESS Directive SUPPRESS overrides all the listing directives. The default is NOSUPPRESS.

SYMBOLPAGES Directive SYMBOLPAGES sets the size of the internal symbol table the compiler uses as a temporary storage area for processing variables and SQL statements.



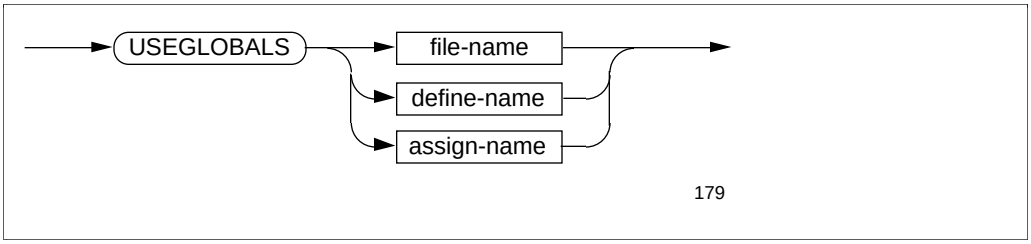
SYMBOLS Directive SYMBOLS saves symbols in a symbol table (for Inspect symbolic debugging) in the object file. The default is NOSYMBOLS.

SYNTAX Directive SYNTAX checks the syntax of the source text without producing an object file.

TARGET Directive See “Privileged Procedures.”

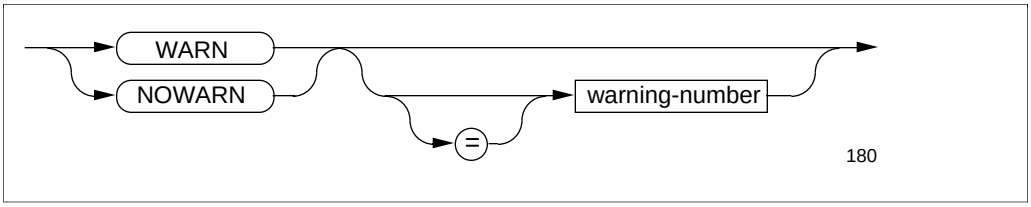
USEGLOBALS Directive

USEGLOBALS retrieves the global data declarations saved in a file by SAVEGLOBALS during a previous compilation.



WARN Directive

WARN instructs the compiler to print a selected warning or all warnings. The default is WARN.



Compiler Initialization Messages

Initialization messages are unnumbered diagnostic messages that can appear during compiler initialization. If the OUT file is available, the compiler sends the messages there; otherwise, it sends them to the home terminal.

An initialization error terminates the compilation with a completion code of 3; that is, the compiler could not access a file. No object file is created.

Initialization messages are:

```
CREATION OF TEMPORARY FILE FAILED

ILLEGAL INPUT DEVICE

LIST DEVICE NOT AVAILABLE

LIST FILE CANNOT BE AN EDIT FILE

OPEN OF INITIAL SOURCE FILE FAILED

OPEN OF TEMPORARY FILE FAILED

REWIND OF SOURCE FILE FAILED
```

Any of these self-explanatory messages can be followed by a line of the format:

```
FILE MANAGEMENT ERROR #error-number ON FILE : file-name
```

About Error and Warning Messages

The compiler scans each line of the source code and notifies you of an error or potential error by displaying one of two types of messages:

Message	Meaning	User Action
Error	Indicates a source error that prevents compilation of the source file into an object file.	Correct the error and recompile the source code.
Warning	Indicates a potential error condition that might affect program compilation or execution.	Check the source code carefully. If your program is adversely affected, make corrections and recompile the source code.

To indicate the location of the error or potential error, the compiler prints a circumflex symbol (^) in the source listing. The circumflex usually appears under the first character position following the detection of the error. (However, if the error involves the relationship of the current source line with a previous line, the circumflex does not always point to the actual error.)

On the next line, the compiler displays a message describing the nature of the error. The forms of error and warning messages are:

```
**** ERROR **** message-number -- message-text
**** WARNING **** message-number -- message-text
```

Occasionally, the compiler adds a third line for supplemental information. For example, the following message refers you to an earlier procedure that contains an error:

```
IN PROC proc-name
```

As another example, the following line refers you to a previous page that contains an error:

```
PREVIOUS ON PAGE #page-num
```

Error messages are described on the following pages in ascending numeric order, followed by warning messages. Although the compiler prints each message on a single line, some messages here are continued on additional lines because of line limitations.

Messages no longer in use are not shown in the list. Thus, a few numbers are omitted from the numeric sequence.

Error Messages

Error diagnostic messages identify source errors that prevent correct compilation. No object file is produced for the compilation.

0 `Compiler error`

This error means that the compiler's data is no longer correct.

If the IN and OUT file numbers are incorrect when the compiler tries to send error 0 to the OUT file, the following file error appears at the home terminal. This error means the file has not been opened.

```
??: 016
```

Error 0 can occur after syntax errors or after a logic error. Error 0 is sometimes preceded by other error messages.

- Syntax error. Correct all syntax errors and recompile. If error 0 persists, contact your Tandem representative. The following syntax errors, for example, cause error 0:
  - A substructure that has an odd length, starts with an INT item, and has the same nonzero upper and lower bounds. The compiler does not pad this substructure properly.
  - Syntax errors within a structure declaration.
  - Nesting of procedure declarations or the appearance of such nesting, such as a formal parameter declared as a procedure but not included in the formal parameter list.
  - Parameter identifiers not separated by commas in the formal parameter list of a procedure declaration.
  - A formal parameter identifier that is a reserved keyword.
  - A BLOCK or NAME identifier that appears outside its declaration.
  - Too many actual parameters in a dynamic procedure call.
  - DEFINE text that begins with, but does not close with, a single quotation mark.
  - An invalid conditional expression in an IF statement or IF expression.
  - An invalid @ on the left side of an assignment.
  - Use of FILLER or BIT_FILLER as variable identifiers within a structure.
- Logic error in compiler operation. A number follows the message for the use of Tandem development personnel. Report this occurrence to Tandem and include a copy of the complete compilation listing (and source, if possible).

1 `Parameter mismatch`

A parameter mismatch, such as the following, has occurred:

- The parameter type of an actual parameter is not the parameter type expected by that procedure. Pass an actual parameter that has the expected parameter type.
- The addressing mode (standard versus extended) of a parameter declaration does not match the addressing mode of its FORWARD or EXTERNAL declaration. Correct the addressing mode in the parameter declaration to match its FORWARD or EXTERNAL declaration.

2 `Identifier declared more than once`

A declaration contains an identifier that is already declared within this scope as shown in the following example. Replace the identifier with one that is unique within this scope.

```
PROC p;
  BEGIN
  LABEL result;
  INT   result;           !Duplicate identifier
  END;
```

3 `Recursive DEFINE invocation`

A reference to a DEFINE declaration that is recursive appears in your source code. The message appears when the compiler expands the DEFINE. In the following example, the compiler expands A, which expands B, which expands A, and so on:

```
DEFINE a = b#, b = a#;
```

Rewrite the DEFINE so that it does not call itself.

4 `Illegal MOVE statement or group comparison`

An incorrect move statement or group comparison expression appears, for which the compiler cannot generate code. Correct the statement or expression.

5

```
Global primary area exceeds 256 words
```

The space required for your global variables exceeds the 256-word global primary area. The compiler allocates the following kinds of global variables in this area:

- ☐ Directly addressed simple variables, arrays, and structures
- ☐ 16-bit or 32-bit pointers you declare and initialize yourself
- ☐ 16-bit or 32-bit implicit pointers for indirect arrays and indirect structures
- ☐ Two 32-bit extended-stack pointers if you declared extended local variables

Declare most global arrays and structures by using indirection. Declare very large global arrays and structures by using extended indirection.

6

```
Illegal digit
```

A numeric constant contains a digit that is incorrect in the stated base of the constant. For example, an octal constant contains the digit 9. Correct the constant in accordance with Section 3, “Data Representation.”

7

```
String overflow
```

A character string appears that:

- ☐ Contains more than 128 characters. Reduce the length of the character string.
- ☐ Does not terminate in the line in which it begins. Specify a constant list of smaller character strings; for example:

```
STRING a[0:99] := ["These two constant strings will "
                   "appear as if they were one character string."]
```

8

```
Not defined for INT(32), FIXED, or REAL
```

An arithmetic operation occurs that is not permissible for operands of the listed data types. Correct the expression in accordance with Section 4, “Expressions.”

9 `Compiler does not allocate space for .SG STRUCTs`

A structure declaration incorrectly includes .SG (the system global indirection symbol). To access the system global area, you can:

- ☐ Equivalence a structure to a location relative to the base address of the system global area. (See Section 15, “Privileged Procedures.”)
- ☐ Declare a system global pointer (using the .SG symbol) and assign a structure address to it.

Otherwise, declare a global or local structure using the standard or extended indirection symbol (. or .EXT) and declare a sublocal structure without using any indirection symbol.

10 `Address range violation`

This message indicates one of the following conditions:

- ☐ A declaration specifies addresses beyond the allowable range; for example:

  ```
  INT i = 'G' + 300;
  ```

  Declare at most 256 words of directly addressable data relative to 'G', 127 words relative to 'L', and 31 words relative to 'S'.
- ☐ A subprocedure parameter or a sublocal variable cannot be accessed because other items have been pushed onto the data stack. Reorder the parameters or sublocal data. “Sublocal Storage Limitations” and “Sublocal Parameter Storage Limitations” in the *TAL Programmer’s Guide* describes the limitations of sublocal storage.
- ☐ The total of primary and secondary global variables exceeds 32K words. Reduce the number or size of your global variables or move some of them to extended memory.
- ☐ The zeroth element of a global direct array is outside G-plus addressing. Declare the array so that its zeroth element falls within G-plus addressing. If the array is located at G[0], its lower bound must be a zero or negative value.
- ☐ The upper bound of the last sublocal array is less than zero. Specify an upper bound that is equal to or larger than zero.
- ☐ The size of an indirect array (of structures) exceeds 32K words. Reduce the size of the array.
- ☐ The size of a local variable exceeds 128K bytes. Reduce the size of the local variable.
- ☐ A procedure that you compile with the SQL directive has more than 122 words of primary local variables. Declare the excess words as indirect variables.

11

```
Illegal reference [ variable-name    ]
                  [ parameter-number ]
```

Conditions that cause this error include:

- A variable appears where a constant is expected, or a constant appears where a variable is expected. Replace *variable-name* with a constant or variable as required.
- A CALL statement passes a value parameter to a procedure that expects a reference parameter. Replace the parameter indicated by *parameter-number* with a reference parameter.
- A CODE statement includes indirect branches such as the following. Remove any indirect branches from the CODE statement.

```
CODE (BANZ .test1);
!Some code here
test1: CODE (CON @test2);
```

12

```
Nested routine declaration(s)
```

The following conditions can cause this error:

- One or more procedure declarations appear within another procedure declaration. Either replace the nested procedures with subprocedures or move the nested procedures outside the encompassing procedure.
- The identifier of a procedure declared as a parameter does not appear in the parameter list. Add the procedure identifier to the parameter list. For example:

```
PROC myproc (a);    !Q is missing from parameter list
    INT a;
    PROC q;         !Q is declared as parameter
  BEGIN
  !Lots of code
  END;
```

13

```
Only 16-bit INT value(s) allowed
```

- A value of a data type other than INT appears where only INT values are permitted. Specify INT values in this context.
- An index of a data type other than INT appears for an equivalenced address. Specify an INT index for the equivalenced address.

14 `Only initialization with constant value(s) is allowed`

A global initialization expression includes variables. Initialize global data only with constant expressions, which can include *@identifier* when used with standard functions that return a constant value. @ accesses the address of *identifier*, which must be a nonpointer variable with a 16-bit address. For example:

```
STRUCT .st[0:1];
BEGIN
  INT a;
  INT b;
  INT c;
END;

INT .p1 := @st.b;                          -- error 14
INT .p2 := @st '+' $OFFSET (st.b) / 2;     -- works fine

INT .q1 := @st[1].c;                       -- error 14
INT .q2 := @st '+' $LEN (st) / 2
               '+' $OFFSET (st.c) / 2;     -- works fine
```

15 `Initialization is illegal with reference specification`

An equivalenced variable declaration tries to initialize the previously declared variable. Initialize the previous variable before referring to it in the equivalenced variable declaration. For example:

```
INT b;
INT .a = b := 5;        !Cannot initialize B here; error 15

INT b := 5;             !Can initialize B here
INT .a = b;             !No error
```

16 `Insufficient disk space on swap volume`

The swap volume cannot accommodate your program. For example, either of the following values are too large for the memory available on the swap volume:

- ☐ The SQL PAGES value
- ☐ The combined SQL PAGES and SYMBOLPAGES values

Reduce the value that is too large, or specify another swap volume.

17 `Formal parameter type specification is missing`

A declaration for a formal parameter is missing in the procedure or subprocedure header. Declare the missing formal parameter or remove its identifier from the formal parameter list.

18 `Illegal array bounds specification`

Incorrect bounds appear in an array declaration. To correct this error:

- ☐ Specify bounds that are constant expressions.
- ☐ Specify a lower bound that is smaller than the upper bound. (This is not a requirement when the array is declared within a structure.)
- ☐ Specify no bounds when you declare an equivalenced variable:

```
INT a[0:5] = b;       !Cause error 18
```

19 `Global or nested SUBPROC declaration`

A subprocedure declaration appears either outside a procedure or within another subprocedure. Declare all subprocedures within a procedure but not within a subprocedure.

20 `Illegal bit field designator`

A bit field construct is incorrect. Correct the construct so that both bit numbers are INT constants and the left bit number is less than or equal to the right bit number; for example:

```
<0:5>
```

21 `Label declared more than once`

Duplicate label identifiers appear in the same procedure as shown in the following example. Specify label identifiers that are unique within a procedure.

```
PROC q;
  BEGIN
  mylabel:
    !Some statements here
  mylabel:                  !Duplicate label identifier
    !More statements here
  END;
```

22 `Only standard indirect variables are allowed`

The extended (32-bit) indirection symbol (.EXT) appears where the compiler expects standard indirection. Use the standard (16-bit) indirection symbol (.) in this context.

23 `Variable size error`

The size field of a data type is incorrect. For example, INT(12) is incorrect. Specify a correct data type, such as INT or INT(32).

24 `Data declaration(s) must precede PROC declaration(s)`

A global data declaration follows a procedure declaration. Declare all global data before the first procedure declaration.

25 `Item does not have an extended address`

The argument to the standard function $LADR does not have an extended address. When you use $LADR, specify an argument that has an extended address.

26 `Routine declared forward more than once`

More than one FORWARD declaration for the given procedure or subprocedure is present. Declare a procedure or subprocedure FORWARD only once. Delete all duplicate FORWARD declarations.

27 `Illegal syntax`

A statement or the line preceding it contains one or more syntax errors. For example, the following conditions can cause error 27, followed by error 0, which terminates compilation:

- A misplaced or missing semicolon; for example:

```
PROC myproc          !Place semicolon here
  BEGIN;             !Remove this semicolon
  INT num; sum;      !Replace first semicolon with comma
  num := 2           !Place semicolon here
  sum := 5;
  END;
```

- A reserved word appears as an identifier in the formal parameter list of a procedure declaration. Replace the identifier with a nonreserved identifier.
- An incorrect conditional expression appears in an IF statement or IF expression. Correct the expression.

28 `Illegal use of code relative variable`

A read-only array appears in an incorrect context, such as:

- On the left side of an assignment operator (:=)
- On the left side of a move operator (':=' or '=:')
- On the left side of a group comparison expression

You can use a normal array instead. For the syntax of arrays, see Section 7.

29 `Illegal use of identifier` *`name`*

An identifier appears in the formal parameter specification of a procedure or subprocedure declaration but not in the formal parameter list. Either include the identifier in the formal parameter list or remove the identifier from the formal parameter specification.

30 `Only label or USE variable allowed`

A DROP statement specifies an incorrect identifier. In the DROP statement, specify the identifier of a label or a USE statement variable.

31 `Only PROC or SUBPROC identifier allowed`

A CALL statement specifies an incorrect identifier. In the CALL statement, specify the identifier of a procedure, subprocedure, or entry point.

32 `Type incompatibility`

This message indicates one of the following conditions:

- An expression contains operands of different data types. To make the types match, use type-transfer standard functions or specify constants correctly. For example, for an INT(32) constant, use the D suffix.
- A procedure without a return type occurs on the right side of an assignment statement. Specify only a function in this context.
- Either a procedure is declared INT and its FORWARD declaration is declared INT(32), or a procedure is declared INT(32) and its FORWARD declaration is declared INT. Make the data type in both declarations match.
- A RETURN statement has no return value and the correct number of words for a return value is not in the register stack. In an INT procedure, for example, specify an INT value in the RETURN statement.
- A RETURN statement specifies a return value of the wrong data type. Specify the correct data type.
- An undeclared variable prefixed by @ is passed as an actual parameter. The compiler treats the undeclared variable as a label, so if the allocated size of the formal parameter does not match that of the label address, the compiler issues the error. Declare the variable before passing it as a parameter.
- A character string is assigned to a FIXED or REAL(64) variable. Change the value from a character string, or change the data type.

33 `Illegal global declaration(s)`

A declaration occurs for an item (such as a label) that cannot be a global item. At the global level, you can declare LITERALs, DEFINEs, simple variables, arrays, structures, simple pointers, structure pointers, and equivalenced variables.

34 `Missing variable`

A required variable is missing from the current statement. Specify all variables shown as required in the syntax diagrams in this manual.

35 `Subprocedures cannot be parameters`

A subprocedure is declared as a formal parameter or is passed as an actual parameter. You can declare and pass procedures (but not subprocedures) as parameters.

36 `Illegal range`

A specified value exceeds the allowable range for a given operation. Correct the value.

37 `Missing identifier`

A required identifier is missing from the current statement. Provide the missing identifier.

38 `Illegal index register specification`

You tried to reserve more than three registers for use as index registers. Use a DROP statement to reduce the number of reserved registers.

39 `Open failed on file` *`file-name`*

The compiler could not open the file you specified in a SOURCE directive. If NOABORT is in effect, the compiler prompts you for the name of a source file. You can take any of the following alternative actions:

- ☐ Retry the same source file name.
- ☐ Ignore that source file and continue compilation.
- ☐ Substitute another source file name.
- ☐ Terminate the compilation.

If you choose to ignore the source file or to terminate the compilation, error 39 appears. If SUPPRESS is in effect, the compiler prints the SOURCE directive before the error message.

40 `Only allowed with a variable`

- A bit-deposit construct is applied to a variable other than STRING or INT:

```
INT i;
UNSIGNED(5) uns5;

i := uns5.<13:14>;        !Error 40 appears
```

To correct the preceding error, change the variable to STRING or INT:

```
INT i, var;

i := var.<13:14>;
```

- An operation or construct that is valid only when used with a variable appears in some other context, such as appending a bit-deposit field to an expression:

```
INT a, b;

(a+b).<2:5> := 0;         !Error 40 appears
```

To correct the preceding error, assign the expression to a STRING or INT variable and then append the bit-deposit field to the variable:

```
INT a, b, var;

!Code to store values in A and B
var := a + b;
var.<2:5> := 0;
```

41 `Undefined ASSERTION procedure: proc-name`

An ASSERT statement invokes an undeclared procedure specified in an ASSERTION directive. Either declare the procedure or specify an existing procedure in the ASSERTION directive.

42

```
Table overflow number
```

Your source program fills one of the fixed-size tables of the compiler. No recovery from this condition is possible. Correct the source program as indicated in the following table (*number* identifies the affected table):

Number	Table Name	Condition/Action
0	Constant	Place a DUMPCONS directive before the point of overflow to force the constant table to be dumped. Termination does not occur if a block move of a large constant list caused the overflow.
1	Tree	Simplify the expression.
2	Pseudo-Label	You might have too many nested IF statements. Simplify the IF statements.
3	Parametric DEFINE	The DEFINE macro being expanded has parameters that are too long. Shorten the parameters.
4	Section	SOURCE directives access too many sections at one time. Break the sections into two or more groups.

43

```
Illegal symbol
```

The current source line contains an invalid character or a character that is invalid in the current context. Specify the correct character.

44

```
Illegal instruction
```

The specified mnemonic does not match those for the NonStop system. As of release C00, the compiler does not generate code for the NonStop 1+ system. Replace the instruction with one described in the *System Description Manual* for your system.

45

```
Only INT(32) value(s) allowed
```

A value of the wrong data type appears. In this context, specify an INT(32) value.

46

```
Illegal indirection specification
```

The period symbol (.) is used on a variable that is already indirect. Specify only one level of indirection by removing the period symbol from the current context.

47 `Illegal for 16-bit INT`

An INT value appears where the compiler expected an INT(32) value. For the unsigned divide ('/') and unsigned modulo divide ('\') operations, specify an INT(32) dividend and an INT divisor.

48 `Missing item-specification`

The source code is missing *item-specification*. Supply the missing item.

49 `Undeclared identifier`

- ☐ A reference to an undeclared data item appears in the source file. Declare the item or change the reference.
- ☐ The string parameter of a parameter pair (*string:length*) is misspelled. Correct the spelling.

50 `Cannot drop this Label`

A DROP statement refers to an undeclared or unused label. Drop a label only after you declare it and after the compiler reads all references to it. Dropping a label saves symbol table space and allows its reuse (as in a DEFINE macro).

51 `Index register allocation failed`

The compiler is unable to allocate an index register. You might have indexed multiple arrays in a single statement and reserved the limit of index registers using USE statements. Modify your program so that it requires no more than three index registers at a time.

52 `Missing initialization for code relative array`

Initialization is missing from a read-only array declaration. When you declare a read-only array, make sure you initialize the array with values (not including "").

53 `Edit file has invalid format or sequence n`

The compiler detects an unrecoverable error in the source file. In the message, *n* is a negative number that identifies one of the following conditions:

Number	Condition/Action
–3	Text-file format error. Correct the format.
–4	Sequence error—the line number of the current source line is less than that of the preceding line. Correct the sequence of source lines.

54 `Illegal reference parameter`

A structure appears as a formal value parameter. Declare all structure formal parameters as reference parameters.

55 `Illegal SUBPROC attribute`

A subprocedure declaration includes an incorrect subprocedure attribute such as EXTERNAL or EXTENSIBLE. Remove the incorrect attribute from the subprocedure declaration. Subprocedures can only have the VARIABLE attribute.

56 `Illegal use of USE variable`

A USE variable is used incorrectly. Correct the usage in accordance with the descriptions of the USE statement or the optimized FOR statement.

57 `Symbol table overflow`

The compiler has not allocated sufficient space for the symbols in your program. You can either:

- ☐ Use the SYMBOLPAGES directive to increase the allocation
- ☐ Break the program into smaller modules

58 `Illegal branch`

Your program branches into a FOR statement that uses a USE register as its counter. Branch to the beginning of the FOR statement, not within it.

59 `Division by zero`

The compiler detects an attempt to divide by 0. Correct the expression to avoid division by 0.

60 `Only a data variable may be indexed`

An index is appended to an invalid identifier such as the identifier of a label or an entry point. Append an index only to the identifier of a variable.

61 `Actual/formal parameter count mismatch`

A call to a procedure or subprocedure supplies more (or fewer) parameters than you defined in the procedure or subprocedure declaration. Supply all required parameters, and supply at least commas for all optional parameters.

62 `Forward/external parameter count mismatch`

The number of parameters specified in a FORWARD or EXTERNAL declaration differs from that specified in the procedure body declaration. Specify the same number of parameters in the procedure body declaration as there are in the FORWARD or EXTERNAL declaration.

63 `Illegal drop of USE variable in context of FOR loop`

Within a FOR loop, a DROP statement attempts to drop a USE register that is the index of the FOR loop. The FOR loop can function correctly only if the register remains reserved. Remove the DROP statement from within the FOR loop.

64 `Scale point must be a constant`

The current source line contains a scale point that is not a constant. Specify the *fpoint* in a FIXED variable declaration and the *scale* argument to the $SCALE function as INT constants in the range -19 through +19.

65 `Illegal parameter or routine not variable`

The *formal-param* supplied to the $PARAM function is not in the formal parameter list for the procedure, or the $PARAM function appears in a procedure that is not VARIABLE or EXTENSIBLE. Use the $PARAM function only in VARIABLE procedures and subprocedures and in EXTENSIBLE procedures.

66 `Unable to process remaining text`

This message is usually the result of a poorly structured program, when numerous errors are compounded and concatenated to the point where the compiler is unable to proceed with the analysis of the remaining source lines. Review the code to determine how to correct the errors.

67 `Source commands nested too deeply`

The compilation unit nests SOURCE directives to more than seven levels, not counting the original outermost source file. That is, a SOURCE directive reads in source code that contains a SOURCE directive that reads in source code that contains a SOURCE directive that reads in source code, and so on, until the seven-level limit is exceeded. Reduce the number of nested levels.

68 `This identifier cannot be indexed`

A directly addressable variable was indexed and used in a memory-referencing instruction in a CODE statement. Modify the code to avoid this usage.

69 `Invalid template access`

A template structure is referenced as an allocated data item, such as in the $OCCURS function. Refer to a template structure only in the declaration of a referral structure or a structure pointer.

70 `Only items subordinate to a structure may be qualified`

A qualified reference appears for a nonstructure item. Use the qualified identifier form of *structure-name.substructure-name.item-name* only for data items within a structure.

71

```
Only INT or STRING STRUCT pointers are allowed
```

A structure pointer of an incorrect data type occurs. Specify only INT or STRING data type when you declare structure pointers.

72

```
Indirection must be supplied
```

An indirection symbol is missing from a pointer declaration. When you declare a pointer, specify an indirection symbol preceding the pointer identifier.

73

```
Only a structure identifier may be used as a referral
```

An incorrect referral occurs in a declaration. For the referral, specify the identifier of a previously declared structure or structure pointer.

74

```
Word addressable items may not be accessed through a
STRING structure pointer
```

A STRING structure pointer attempts to access word-addressed items. To access word-addressed structure items, use an INT structure pointer, either a standard (16-bit) pointer or an extended (32-bit) pointer.

75

```
Illegal UNSIGNED variable declaration
```

An indirect UNSIGNED variable declaration occurs. Declare an UNSIGNED variable as directly addressed, regardless of its scope (global, local, or sublocal).

76

```
Illegal STRUCT or SUBSTRUCT reference
```

An incorrect structure or substructure reference occurs. Refer to a structure or substructure only:

- ☐ In a move statement
- ☐ In a group comparison expression
- ☐ In a SCAN or RSCAN statement
- ☐ As an actual reference parameter
- ☐ As *@identifier* in an expression

77 `Unsigned variables may not be subscripted`

An indexed (subscripted) reference to an UNSIGNED simple variable occurs. Remove the index from the identifier of the UNSIGNED simple variable.

78 `Invalid number form`

A floating-point constant appears in an incorrect form. Use one of the forms described in "REAL and REAL(64) Numeric Constant" in Section 3, "Data Representation."

79 `REAL underflow or overflow`

Underflow or overflow occurred during input conversion of a REAL or REAL(64) number. Specify floating-point numbers in the following approximate range:

±8.6361685550944446E–78 through ±1.15792089237316189E77

80 `OPTIMIZE 2 register allocation conflict`

The compiler has run out of registers in OPTIMIZE 2 mode. The compiler emits error 80 and continues the compilation.

81 `Invoked forward PROC converted to external`

An EXTERNAL procedure declaration occurs for a procedure that was previously called as if it were a FORWARD procedure. Correct either the EXTERNAL declaration or the call to the FORWARD procedure.

82 `CROSSREF does not work with USEGLOBALS`

The USEGLOBALS directive appears in a source file submitted to the stand-alone Crossref product. The compiler issues error 82 and stops the Crossref product. Before resubmitting the source file to the Crossref product, remove the USEGLOBALS directive from the source file.

83 `CPU type must be set initially`

A CPU directive appears in the wrong place. Specify this directive preceding any data or procedure declarations.

84 `There is no SCAN instruction for extended memory`

An extended indirect array is the object of a SCAN or RSCAN statement. The hardware does not support scans in extended memory. Move the array temporarily into a location in the user data segment and perform the scan operations there.

85 `Bounds illegal on .SG or .EXT items`

A system global or extended pointer declaration includes bound specifications. Remove the bounds from such declarations.

86 `Constant expected and not found`

A variable appears where the compiler expects a constant. Replace the variable with a constant.

87 `Illegal constant format`

A constant appears in an incorrect form. Specify constants in the forms described in Section 3, “Data Representation.”

88 `Expression too complex.  Please simplify`

The current expression is too complex. The compiler’s stack overflowed and the compilation terminated. Simplify the expression.

89 `Only arrays of simple data types can be declared read-only`

A structure declaration contains `= 'P'`, which is restricted to read-only array declarations. Either remove `= 'P'` from the structure declaration, or replace the structure declaration with a read-only array declaration.

90 `Invalid object file name - file-name`

The object file name is incorrect. Specify the name of a disk file.

91 `Invalid default volume or subvolume`

The default volume or subvolume in the startup message is incorrect. Correct the volume or subvolume name.

92 `Branch to entry point not allowed`

An entry point is the target of a GOTO statement. Following the entry point, add a label identifier and then specify the label identifier in the GOTO statement.

93 `Previous data block not ended`

A BLOCK or PROC declaration appears before the end of the previous BLOCK declaration. End each BLOCK declaration with the END BLOCK keywords before starting a new BLOCK declaration or the first PROC declaration. This message occurs only if the compilation begins with a NAME declaration.

94 `Declaration must be in a data block`

An unblocked global data declaration appears after a BLOCK declaration. Either place all unblocked global declarations inside BLOCK declarations or place them before the first BLOCK declaration or SOURCE directive that includes a BLOCK. This message occurs only if the compilation begins with a NAME declaration.

95 `Cannot purge file file-name`

A security violation occurs in a SAVEGLOBALS compilation and the compiler cannot purge the existing global data declarations file. Correct the security violation and then recompile.

96

```
Address references between global data blocks not allowed
```

A variable declared in one global data block is initialized with the address of a variable declared in another global data block. Because global data blocks are relocatable, such an initialization is invalid. Include both declarations in the same global data block (or BLOCK declaration). This message occurs only if the compilation begins with a NAME declaration.

97

```
Equivalences between global data blocks not allowed
```

An equivalenced declaration in a global data block refers to a variable declared in another global data block. Place both declarations in the same global data block (or BLOCK declaration). This message occurs only if the compilation unit begins with the NAME declaration.

98

```
Extended arrays are not allowed in subprocedures
```

A declaration for an extended indirect array appears in a subprocedure. Remove the extended indirection symbol (.EXT) from the array declaration. Sublocal variables must be directly addressed.

99

```
Initialization list exceeds space allocated
```

A constant list contains values that exceed the space allocated by the data declaration. List smaller values in the constant list or declare larger variables.

100

```
Nested parametric-DEFINE definition encountered during
expansion
```

Nesting of DEFINE declarations occurs. Remove the DEFINE declaration that is nested within another DEFINE declaration.

101

```
Illegal conversion to EXTENSIBLE
```

An attempt to convert an ineligible procedure to EXTENSIBLE occurs. Convert only a VARIABLE procedure that has at least one parameter, at most 16 words of parameters, and all one-word parameters except the last, which can be a word or longer. Also specify the number of parameters the procedure had when it was VARIABLE.

102 `Illegal operand for ACON`

A CODE statement contains an incorrect constant for the ACON option. Specify a constant that represents the absolute run-time code address associated with the label in the next instruction location. An absolute code address is relative to the beginning of the code space in which the encompassing procedure resides.

103 `Indirection mode specified not allowed for P-relative variable`

A read-only array declaration includes an indirection symbol. Remove the indirection symbol from the declaration.

104 `This procedure has missing label - label-name`

The procedure refers to either:

- A label identifier that is missing from the procedure
- An undeclared variable prefaced with @ in an actual parameter list

Either use *label-name* in the procedure or declare the variable.

105 `A secondary entry point is missing - entry-point-name`

The entry point is not present in the procedure. Declare an entry-point identifier and use it in the procedure.

106 `A referenced subprocedure declared FORWARD is missing - subproc-name`

The procedure contains a FORWARD subprocedure declaration and a call to that subprocedure but the subprocedure body is missing. Declare the subprocedure body.

108 `Case label must be signed, 16-bit integer`

An incorrect case label appears in a labeled CASE statement. Specify a signed INT constant or a LITERAL for the case label.

109

```
Case label range must be non-empty - range
```

A case alternative in a labeled CASE statement has no values associated with it. Specify at least one value for each case alternative.

110

```
This case label (or range) overlaps a previously used
case label - n
```

The value *n* appears as a case label more than once in a labeled CASE statement. Specify unique case labels within a CASE statement.

111

```
The number of sparse case labels is limited to 63
```

Too many case labels appear in a labeled CASE statement. Specify no more than 63 case labels.

112

```
USEGLOBALS file was created with an old version of TAL or
file code is not 105, file-name
```

The USEGLOBALS directive cannot use the global declarations file named in the message. If the global declarations file does not have file code 105, recompile the source code by using the SAVEGLOBALS directive. Use the same version of the compiler for both the SAVEGLOBALS and USEGLOBALS compilations.

113

```
File error number, file-name
```

A file error occurred when the compiler tried to process a file named in a directive such as ERRORFILE or SAVEGLOBALS. The message includes the name of the file and the number of the file error. Provide the correct file name.

114

```
@ prefix is not allowed on SCAN, MOVE or GROUP COMPARISON
variable
```

A variable in a SCAN or move statement, or in a group comparison expression, is prefixed with @, which returns the address of the variable. Remove the @ operator.

115 `Missing FOR part`

The FOR *count* clause is missing from a move statement. Include the FOR *count* clause in the move statement.

116 `Illegal use of period prefix`

An incorrect use of the period (.) prefix occurs. Use this prefix only as follows:

- As an indirection symbol in declarations
- As a separator in qualified identifiers of structure items, as in MYSTRUCT.SUBSTRUCT.ITEMX
- As a dereferencing symbol with INT and STRING identifiers to add a level of indirection

117 `Bounds are illegal on struct pointers`

A structure pointer declaration includes bounds. Remove the bounds from the declaration.

118 `Width of UNSIGNED array elements must be 1, 2, 4, or 8 bits`

An incorrect *width* value appears in an UNSIGNED array declaration. For the *width* of UNSIGNED array elements, specify 1, 2, 4, or 8 only.

119 `Illegal use of @ prefix together with index expression`

In an assignment statement, an indexed pointer identifier appears. If the pointer is declared within a structure or substructure, append the index to the structure or substructure identifier, not to the pointer identifier. Otherwise, remove the index from the assignment statement.

120 `Only type INT and INT(32) index expressions are allowed`

An index expression of the wrong data type appears. Specify an INT or INT(32) index expression.

121

```
Only STRUCT items are allowed here
```

An incorrect argument to the $BITOFFSET function appears. Specify only a structure item as the $BITOFFSET argument.

122

```
Extended MOVE or GROUP COMPARISON needs a 32-bit NEXT
ADDRESS variable
```

The next-address (*next-addr*) variable in a move statement or a group comparison expression is the wrong data type. Specify an INT(32) *next-addr* variable because the compiler emits an extended move sequence when:

- ☐ The destination variable or the source variable has extended addressing.
- ☐ The destination variable or the source variable has byte addressing and the other has word addressing.

123

```
Not allowed with UNSIGNED variables
```

An incorrect reference to an UNSIGNED variable appears. Correct the reference so that the UNSIGNED identifier:

- ☐ Has no @ or period (.) prefix
- ☐ Is not sent as an actual parameter to a reference formal parameter
- ☐ Is not the source or destination of a move statement, a SCAN or RSCAN statement, or a group comparison expression

124

```
ERRORFILE exists and its file code is not 106, will not
purge
```

The file specified in an ERRORFILE declarative has the wrong file code. The compiler does not purge the file. Specify a file that has file code 106.

125

```
MOVE or GROUP COMPARISON count-unit must be BYTES, WORDS,
or ELEMENTS
```

An incorrect *count-unit* appears in a move statement or in a group comparison expression. Specify the *count-unit* as BYTES, WORDS, or ELEMENTS only.

126

```
Initialization of local extended arrays is not allowed
```

An initial value appears in a local extended array declaration. Remove the initialization from the declaration and use an assignment statement instead.

127

```
Illegal block relocation specifier, expecting either AT
or BELOW
```

An incorrect relocation clause appears in a BLOCK declaration. Specify only the AT or BELOW clause to relocate the block.

128

```
Illegal block relocation position
```

An incorrect relocation position appears in a BLOCK declaration. Specify AT (0), BELOW (64), or BELOW (256) only.

129

```
This variable must be subscripted
```

A reference to an UNSIGNED array appears without an index (subscript). When you refer to an UNSIGNED array, always append an index to the array identifier.

130

```
This variable may not be the target of an equivalence
```

The new variable in an equivalenced variable declaration is type UNSIGNED. Declare the new variable as any type except UNSIGNED.

131

```
Procedure code space exceeds 32K words
```

A single procedure is larger than 32K words long. Reduce the size of the procedure.

132

```
Compiler internal logic error detected in code generator
```

The code generator detected an internal logic error. Report this occurrence to Tandem and include a copy of the complete compilation listing (and source, if possible).

133

```
Compiler label table overflow
```

The code generator detected a label table overflow. Report this occurrence to Tandem and include a copy of the complete compilation listing (and source, if possible).

134

```
Value assigned to USE variable within argument list
may be lost
```

An assignment to a USE register appears in an actual parameter list. After a procedure call, the compiler restores the USE register to its original value and the changed value is lost. Before issuing the CALL statement, use an assignment statement to change the value in the USE register.

135

```
Use relational expression
```

UNSIGNED(17–31) operands appear incorrectly in a conditional expression. To correct the expression, either change the data type or change the operator. With Boolean operators and unsigned relational operators, use only STRING, INT, or UNSIGNED(1–16) operands. With signed relational operators, use operands of any data type, including UNSIGNED(17–31) operands.

136

```
Compiler relative reference table overflow
```

The compiler's relative reference table overflowed. Report this occurrence to Tandem and include a copy of the complete compilation listing (and source, if possible).

137

```
Cannot purge error file file-name - File system error number
```

The compiler encountered a file error when it tried to purge an error file whose name was specified in an ERRORFILE directive. The message includes the name of the file and the number of the file system error. Supply the correct file name.

138

```
Not a host variable
```

A data item appears where an SQL host variable is expected. Declare the data item in an EXEC SQL DECLARE block, as described in the *NonStop SQL Programming Manual for TAL*.

139 `Invalid declaration for length component of string parameter`

An incorrect *length* parameter appears in a parameter pair specification. Declare this parameter as an INT simple variable that specifies the length of the *string* parameter in the parameter pair.

140 `Too many parameters`

Too many formal parameters appear in a procedure or subprocedure declaration. For a procedure, include no more than 32 formal parameters. For a subprocedure, include no more than allowed by the space available in the parameter area.

141 `Invalid declaration for string component of string parameter`

An incorrect *string* parameter appears in a parameter pair specification of the form *string:length*. Declare the *string* parameter as a standard indirect or extended indirect STRING array.

142 `String parameter pair expected`

A CALL statement passes an actual parameter to a procedure that expects a parameter pair. In the actual parameter list, replace the incorrect parameter with a parameter pair in the form: *string: length*

143 `String parameter pair not expected`

A CALL statement passes a parameter pair to a procedure that does not expect it. In the actual parameter list, replace the parameter pair with a single parameter.

144 `Colon not allowed in the actual parameter list`

A colon appears incorrectly in an actual parameter list. If the colon represents an omitted actual parameter pair, replace the colon with a comma. Use a colon only between the *string* and *length* parameters of a parameter pair.

145

```
Only 16-bit integer index expression allowed
```

An index expression of the wrong data type appears. Change the index expression to an INT expression.

146

```
Identifier for SQLMEM length must be an INT literal
```

An incorrect MAPPED length value appears in the SQLMEM directive. Specify the length value as an INT LITERAL or constant. The LITERAL identifier is interpreted when an EXEC SQL statement occurs, not when the directive occurs.

147

```
Identifier for SQLMEM address must be an extended string
pointer
```

An incorrect MAPPED address value appears in the SQLMEM directive. Specify the address value as a constant or an extended indirect identifier of type STRING. The LITERAL identifier is interpreted when an EXEC SQL statement occurs, not when the directive occurs.

148

```
Exceeded allocated space for SQLMEM MAPPED
```

SQL data structures need more space than is allocated. Specify a larger MAPPED length value in the SQLMEM directive.

149

```
Value out of range
```

The specified value is outside the permissible range of values. For example, the value 256 is outside the range for a BIT_FILLER field, which has a range of 0 through 255. Specify a value that falls within the range.

150

```
SQLMEM STACK cannot be used in a SUBPROC
```

Within a subprocedure, the SQLMEM STACK directive is in effect when an SQL statement occurs. Because of addressability limits in subprocedures, parameters or data might not be accessible if you push data onto the stack. Remove the SQLMEM STACK directive from within the subprocedure.

151 `Only STRING arrays may have SQL attributes`

SQL attributes are applied to arrays that are not STRING arrays. Apply SQL attributes only to STRING arrays.

152 `Type mismatch for SQL attribute`

An incorrect SQL data type attribute occurs for a TAL variable. Specify an SQL data type attribute that matches the TAL data type of the variable.

153 `Length mismatch for SQL attribute`

The length of the data provided in the SQL attribute does not match the length of the data for the variable. Specify data having a length that conforms to the data type of the variable.

154 `Exceeded available memory limits`

The compiler has exhausted its internal memory resources. Change the swap volume (by using the PARAM SAMECPU command described in Appendix E in the *TAL Programmer's Guide*). If an excessive value appears in the EXTENDTALHEAP, SYMBOLPAGES, or SQL PAGES directive, specify a smaller value.

155 `This directive not allowed in this part of program`

The directive is not appropriate in its current location.

- ☐ If the directive belongs elsewhere, such as in the compiler run command or on the first line of the source file, relocate the directive.
- ☐ If the directive should not appear because of a previous occurrence of this or another directive, remove the incorrect directive.

156 `ASSERTION procedure cannot be VARIABLE or EXTENSIBLE`

A VARIABLE or EXTENSIBLE procedure is specified in an ASSERTION directive. Specify a procedure that has no parameters.

160 `Only one language attribute is allowed`

More than one language attribute appears in a procedure declaration. Specify only one of the following language attributes following the LANGUAGE keyword (and only in a D-series EXTERNAL procedure declaration):

```
C
COBOL
FORTRAN
PASCAL
UNSPECIFIED
```

161 `Language attribute only allowed for external procedures`

A language attribute appears in a procedure declaration that is not specified as being EXTERNAL. Specify LANGUAGE followed by C, COBOL, FORTRAN, PASCAL, or UNSPECIFIED only in D-series EXTERNAL procedure declarations.

162 `Illegal size given in procedure parameter declaration`

An incorrect parameter declaration appears. Specify the correct parameter type.

163 `Public name only allowed for external procedures`

A public name appears in a procedure declaration that is not specified as being EXTERNAL. Specify a public name only in D-series EXTERNAL procedure declarations.

164 `Procedure was previously declared in another language`

Multiple EXTERNAL declarations have the same procedure identifier but have different language attributes. Delete the incorrect EXTERNAL declaration.

165 `Procedure was previously declared with a public name`

The previous EXTERNAL procedure declaration includes a public name, and the current procedure declaration is a secondary entry point. Remove the public name from the EXTERNAL procedure heading.

166 `Illegal public name encountered`

An incorrect public name appears. Specify a public name only in a D-series EXTERNAL procedure declaration, using the identifier format of the language in which the external routine is written (C, COBOL85, FORTRAN, Pascal, or TAL).

168 `Use a DUMPCONS directive before the atomic operation`

Constants included in the code make it impossible to specify an atomic operation here. Use the DUMPCONS directive to move the constants out of the way.

169
```
Increase the size of the internal heap of the compiler
by recompiling with the EXTENDTALHEAP directive
```

The internal heap of the compiler is too small. Specify the EXTENDTALHEAP directive in the compilation command in all subsequent compilations. An example is:

```
TAL /in mysrc/ myobj; EXTENDTALHEAP 120
```

175
```
$OPTIONAL is only allowed as an actual parameter or
parameter pair
```

An incorrect item appears as an argument to $OPTIONAL. On the right side of the comma in the argument list to $OPTIONAL, specify only an actual parameter or an actual parameter pair. For example:

```
PROC p1 (str:len, b) EXTENSIBLE;
    STRING .str;
    INT len;
    INT b;
  BEGIN
  !Lots of code
  END;

PROC p2;
  BEGIN
  STRING .s[0:79];
  INT i:= 1;
  INT j:= 1;
  CALL p1 ($OPTIONAL (i < 9, s:i),   !Parameter pair
           $OPTIONAL (j > 2, j) );   !Parameter
  END;
```

176 `The second argument of $OPTIONAL must be a string parameter pair`

The called procedure declares a parameter pair, but the caller does not specify a parameter pair as the second argument to $OPTIONAL. Replace the incorrect argument with a parameter pair to match the specification in the called procedure. See the example shown for Error 175.

177 `The first argument of $OPTIONAL must be a 16-bit integer expression`

A conditional expression that does not evaluate to an INT expression appears as the first argument to $OPTIONAL. Correct the conditional expression so that it evaluates to an INT expression. See the example shown for Error 175.

178 `$OPTIONAL allowed only in calls to VARIABLE or EXTENSIBLE procedures`

$OPTIONAL appears in a call to a procedure that is not VARIABLE or EXTENSIBLE. Remove $OPTIONAL from the CALL statement or declare the called procedure as VARIABLE or EXTENSIBLE. See the example shown for Error 175.

179 `Undefined toggle:` *`toggle-name`*

toggle-name is used before it is created. Create the named toggle in a DEFINETOG, RESETTOG, or SETTOG directive before using the toggle in an IF, IFNOT, or ENDIF directive.

Warning Messages

The following messages indicate conditions that might affect program compilation or execution. If you get any warning messages, check your code carefully to determine whether you need to make corrections.

0

```
All index registers are reserved
```

Three index registers are already reserved by USE statements. Three is the maximum number of index registers that you can reserve. The compiler assigned the last identifier to an already allocated index register.

1

```
Identifier exceeds 31 characters in length
```

An identifier in the current source line is longer than 31 characters, the maximum allowed for an identifier. The compiler ignores all excess characters. Make sure the shortened identifier is still unique within its scope.

2

```
Illegal option syntax
```

An incorrect compiler directive option appears. The compiler ignores the option. If omitting the option adversely affects the program, specify the correct option.

3

```
Initialization list exceeds space allocated
```

An initialization list contains more values or characters than can be contained by the variable being initialized. The compiler ignores the excess items. If omitting the excess values adversely affects the program, declare a variable large enough to hold the values.

4

```
P-relative array passed as reference parameter
```

A procedure call passed the address of a read-only array to a procedure. Make sure the procedure takes explicit action to use the address properly.

5

```
PEP size estimate was too small
```

Your PEP estimate (in the PEP directive) is not large enough to contain all the entries required. BINSERV has allocated appropriate additional space.

6 `Invalid ABSLIST addresses may have been generated`

When the file reaches the 64K-word limit, the compiler disables ABSLIST, starts printing offsets relative to the procedure base instead of to the code area base, and emits this warning. Also, because the compiler is a one-pass compiler, some addresses are incorrect when:

- ☐ The file contains more than 32K words of code
- ☐ RESIDENT procedures follow nonresident procedures
- ☐ The PEP directive does not supply enough PEP table space
- ☐ All procedures are not FORWARD (and no PEP directive appears)

If the file has more than 32K words of code space or if you use the stand-alone Binder, do not use ABSLIST.

7 `Multiply defined SECTION` *name*

The same section name appears more than once in the same SOURCE directive. The compiler ignores all occurrences but the first. If omitting the code denoted by duplicate section names adversely affects the program, replace the duplicate section names with unique section names.

8 `SECTION name not found`

A section name listed on a SOURCE directive is not in the specified file. If omitting such code adversely affects the program, specify the correct file name.

9 `RP or S register mismatch`

An operation contains conflicting instructions for the RP (register pointer) or the S register that the compiler cannot resolve. An example of RP conflict is:

```
IF alpha THEN STACK 1 ELSE STACK 1D;
```

An example of S conflict is the following statement. The setting of the S register depends on the value of ALPHA, which the compiler cannot determine at compile time.

```
IF alpha THEN CODE (ADDS 1) ELSE CODE (ADDS 2);
```

The compiler might detect an RP or S conflict in source code that compiled without problem in releases before C00. If the object code does not execute as intended, recode the source code to eliminate the warning. You might only need to insert an RP or DECS directive.

10 `RP register overflow or underflow`

A calculation produced an index register number that is greater than 7 or less than 0. If this overflow or underflow adversely affects your program, correct the calculation.

11 `Parameter type conflict possible`

An actual parameter does not have the parameter type specified by the formal parameter declaration. For example, a procedure passed the address of a byte-aligned (STRING) extended item as an actual parameter to a procedure that expects the address of a word-aligned item. If the address is not on a word boundary, the system ignores the odd-byte number and accesses the entire word. Make sure the size and alignment of the actual parameter matches the requirements of the called procedure.

12 `Undefined option`

Conditions that cause this warning include:

- ☐ An incorrect option in a directive
- ☐ Stray characters (such as semicolons) at the end of a directive line

In either case, the compiler ignores the directive. Enter the correct option or remove the stray characters.

13 `Value out of range`

A value exceeds the permissible range for its context (for example, a shift count is greater than the number of existing bits). If the value is important to your program, use a value that falls within the permissible range.

14 `Index was truncated`

This warning appears, for example, when you:

- ☐ Equivalence a STRING or INT item to an indexed odd-byte address
- ☐ Equivalence a direct variable equivalent to an indexed indirect variable

The compiler truncates the index; for example:

```
STRING .s[0:4]; INT s1 = s[1];     !Result is INT s1 = s[0]
```

15 `Right shift emitted`

A procedure or subprocedure call passed a byte address as a parameter to a procedure that expects a word address. The compiler converts the byte address to a word address by right-shifting the address. If the STRING item begins on an odd-byte boundary, the word-aligned item also includes the even-byte part of the word. If this is a problem, pass only a word address.

16 `Value passed as reference parameter`

A parameter is passed by value to a procedure or subprocedure that expects a reference parameter. If this is your intent, and if the value can be interpreted as a 16-bit address, no error is involved.

17 `Initialization value too complex`

An initialization expression is too complicated to evaluate in the current context. If your program is adversely affected, simplify the expression.

19 `PROC not declared FORWARD with ABSLIST option on`

A PEP directive or a FORWARD declaration is missing. When you use the ABSLIST directive, the compiler must know the size of the PEP table before the procedure occurs in the source program. Enter either a PEP directive at the beginning of the program or a FORWARD declaration for the procedure. The compiler also emits warning 6 at the end of the compilation.

20 `Source line truncated`

A source line extends beyond 132 characters. The compiler ignores the excess characters. If your program is adversely affected, break the source line into lines of less than 132 characters.

21 `Attribute mismatch`

The attributes in a FORWARD declaration do not match those in the procedure body declaration. If your program is adversely affected, correct the set of attributes.

22

```
Illegal command list format
```

The format of options in a directive is incorrect. The compiler ignores the directive. If your program is adversely affected, correct the format of the compiler options.

23

```
The list length has been used for the compare count
```

A FOR count clause and a constant list both appear in a group comparison expression, which are mutually exclusive. The compiler obtains the count of items from the length of the constant list. If your program is adversely affected, correct the group comparison expression as described in Section 4, “Expressions.”

24

```
A USE register has been overwritten
```

The evaluation of an expression caused the value in a USE register to be overwritten. Multiplication of two FIXED values, for example, can cause this to occur. If this affects your program adversely, use a local variable in place of the USE register.

25

```
FIXED point scaling mismatch
```

The *fpoint* of an actual FIXED reference parameter does not match that of the formal parameter. The system applies the *fpoint* of the formal parameter to the actual parameter.

26

```
Arithmetic overflow
```

A numeric constant represents a value that is too large for its data type, or an overflow occurs while the compiler scales a quadrupleword constant up or down.

27

```
ABS (FPOINT) > (19)
```

The *fpoint* in a FIXED declaration or the *scale* parameter of the $SCALE function is less than –19 or greater than 19. The compiler sets the *fpoint* to the maximum limit, either –19 or 19.

28

```
More than one MAIN specified.  MAIN is still name
```

Although the source code can contain more than one MAIN procedure, in the object code only the first MAIN procedure the compiler sees retains the MAIN attribute.

If the program contains any COBOL or COBOL85 program units, the MAIN procedure must be written in COBOL or COBOL85, respectively. For more information, see the *COBOL85 Reference Manual*.

29

```
One or more illegal attributes
```

Incorrect attributes appear in a subprocedure declaration, which can have only the VARIABLE attribute. The compiler ignores all attributes but VARIABLE.

32

```
RETURN not encountered in typed PROC or SUBPROC
```

A RETURN statement is missing from a function or the RP counter of the compiler is 7 (empty register stack). To return a value from the function, include at least one RETURN statement with an expression; this action automatically places the value on the register stack. You can alternatively use a CODE or STACK statement to place the value on the register stack; however, this practice might be compatible only with TNS systems.

33

```
Redefinition size conflict
```

In a structure declaration, a variable redefinition appears in which the new item is larger than the previously declared item. If your program is adversely affected, correct the variable redefinition so that the new item is the same size or shorter than the previous item.

34

```
Redefinition offset conflict
```

In a structure declaration, a variable redefinition appears in which the new item requires word alignment, while the previously declared item has an odd-byte alignment. If your program is adversely affected, correct the variable redefinition so that the new item and the previous item are both word aligned or both byte aligned.

35 `Segment number information lost`

If you pass an extended indirect actual parameter to a procedure that expects a standard address, the compiler converts the address and the segment number is lost. The resulting address is correct only within the user data segment. Pass an appropriate address.

36 `Expression passed as reference parameter`

A calling sequence passes a parameter in the form *@identifier* to a procedure or subprocedure that expects an extended pointer as a parameter. If the intent is to pass the address of the pointer rather than the address stored in the pointer, no error is involved.

37 `Array access changed from indirect to direct`

An indirect array is declared inside a subprocedure or is declared as a read-only array. The compiler changes the indirect array to a direct array because:

- The sublocal area has no secondary storage for indirect data.
- Code-relative addresses are always direct.

38 `S register underflow`

The compiler attempted to generate code that decrements the S register below its initial value in a procedure or subprocedure. If you decide that your source program is correct, you must insert a DECS directive at that point to recalibrate the compiler's internal S register. This is essential in subprocedures, because sublocal variables have S-relative addresses.

39 `This directive cannot be pushed or popped`

A PUSH or POP prefix appears on a directive that has no directive stack. Specify the directive identifier without the prefix.

40 `A procedure declared FORWARD is missing - proc-name`

A FORWARD declaration occurs, but the procedure body declaration is missing. The compiler converts all references to this procedure into EXTERNAL references to the same identifier. If this is not your intent, declare the procedure body.

41 `File system DEFINEs are not enabled`

A TACL DEFINE name appears in place of a file name, but the system is not configured to use TACL DEFINEs. Error 39 (Open failed on *file-name*) follows warning 41. Issue the TACL commands that enable the TACL DEFINE.

42 `Specified bit extract/deposit may be invalid for strings`

An incorrect bit specification occurs. To access or deposit bits in a STRING item, specify bit numbers 8 through 15 only. Specifying bit numbers 0 through 7 of a STRING item has no effect, because the system stores STRING items in the right byte of a word and places a zero in the left byte.

43 `A default OCCURS count of 1 is returned`

$OCCURS appears with an argument that is a simple variable, pointer, or procedure parameter, so OCCURS returns a 1. Use $OCCURS only with an array (declared in or out of a structure), a structure, or a substructure (but not a template structure or a template substructure).

44 `A subprocedure declared FORWARD is missing - subproc-name`

The named subprocedure is declared FORWARD but is not referenced, and its body is not declared. If the absence the body of *subproc-name* adversely affects your program, declare the subprocedure body.

45 `Variable attribute ignored - no parameters`

The VARIABLE attribute appears for a procedure or subprocedure that has no parameters. The compiler ignores the VARIABLE attribute.

46 `Non-relocatable global reference`

A declaration that refers to a G-relative location appears when the RELOCATE directive is in effect and all primary global data is relocatable. This reference might be incorrect if BINSERV relocates the data blocks when it builds the object file. Either change the declaration of the identifier or, if NAME (and BLOCK) statements do not appear, delete the RELOCATE directive.

47 `Invalid file or subvolume specification`

An incorrect file or subvolume appears in a directive. Correct the directive.

48 `This directive not allowed in this part of program`

A directive occurs in an inappropriate place. The compiler ignores the directive. If this adversely affects your program, move the directive to an appropriate location, usually to the first line of the program or to the compilation command.

49 `Address of entry point used in an expression`

The value of the construct *@entry-point-name* for a subprocedure is the address of the first word of code executed after a call to the entry point. If code written for releases before compiler version E01 contains the expression *@ep*-1 to calculate the entry-point location, change it to *@ep* for correct execution.

50 `Literal initialized with address reference`

An incorrect global initialization occurs, in which a LITERAL represents the address of a global variable. Because global data is now relocatable, avoid initializing them with addresses. If your program is adversely affected, initialize the global variable properly.

52

```
The compiler no longer generates code for NonStop 1+ systems
```

As of release C00, the compiler no longer generates code for the NonStop 1+ system. Instructions that apply only to NonStop 1+ systems are CBDA, CBDB, CBUA, CBUB, CDS, RMD, SCAL, SLRU, LWBG, MAPP, MNGS, MNSG, MNSS, UMPP, XINC, XSMS, ZZZZ. A CODE statement that specifies such an instruction causes error 44 (Illegal instruction).

53

```
Illegal order of directives on directive line
```

A directive that must be first on a directive line is not first, or a directive that must be last on a directive line is not last. Place the following directives as indicated below.

Directive	Place in Directive Line	Directive	Place in Directive Line
ASSERTION	Last	IF	Last
COLUMNS	First	SECTION	Alone
ENDIF	Alone	SOURCE	Last

If the COLUMNS directive appears in the compilation command, the compiler does not enforce the above ordering.

54

```
The structure item rather than the define will be referenced
```

A DEFINE declaration renames a structure item, but the qualified identifier of the DEFINE is the same as that of another structure item. A reference to the DEFINE identifier accesses the other structure item. To ensure proper references, use unique identifiers for all declarations.

55

```
The length of this structure exceeds 32767 bytes
at item ** item-name
```

A structure occurrence exceeds 32,767 bytes in length. The message identifies the item that caused the structure to exceed the legal length; the next item is the one the compiler cannot access. Reduce the length of the structure.

57

```
SAVEGLOBALS and USEGLOBALS cannot appear in the same
compilation
```

Two mutually exclusive directives appear in the same compilation unit. To save global declarations in a file for use by a later compilation, retain SAVEGLOBALS and delete USEGLOBALS. To make use of the saved global declarations, retain USEGLOBALS and delete SAVEGLOBALS.

58

```
Code space exceeds 64k, ABSLIST has been disabled
```

TAL compiler versions B00 and later support up to 16 * 64K words of source code. When the code exceeds 64K words, the compiler disables ABSLIST for the remainder of the listing.

59

```
Number of global data blocks exceeds maximum,
SAVEGLOBALS disabled
```

The program attempts to define a global data block beyond the compiler's storage limit of 100 such blocks. Reduce the number of global data blocks.

60

```
Previous errors and warnings will not be included in
ERRORFILE
```

Errors detected and reported before the ERRORFILE directive occurs are not reported to the file specified in the directive.

61

```
This directive can appear only once in a compilation
```

The directive cited in the warning message is not the first of its kind in the compilation. Remove all duplicate occurrences of this directive.

62

```
Extended address of STRING p-rel. array is incorrect if
reference is more than 32K words from array
```

The STRING read-only array is declared in a procedure or subprocedure, and the compiler may generate an incorrect address when converting the address of a global STRING read-only array to an extended address for use in a move statement, a group comparison, or a procedure call.

63

```
A file system DEFINE name is not permitted in a LIBRARY
directive.
```

A TACL ASSIGN name or a TACL DEFINE name appears in a LIBRARY directive. Specify a disk file name, fully or partially qualified, in the LIBRARY directive.

64

```
No file system DEFINE exists for this logical file name
```

A TACL DEFINE name (such as =ABLE) occurs, but no such TACL DEFINE has been added in the operating system environment.

65

```
RELEASE1 and RELEASE2 options are mutually exclusive.
The most recently specified option is in effect.
```

SQL directives specifying both the RELEASE1 and RELEASE2 options appear in the source code. The compiler applies the most recently specified option. If this is not your intent, specify only one of the two options.

66

```
WHENEVERLIST and NOWHENEVERLIST options are mutually
exclusive.  The most recently specified option is in effect.
```

SQL directives specifying the WHENEVERLIST and NOWHENEVERLIST options appear in the source code. The compiler applies the most recently specified option. If this is not your intent, specify only one of the two options.

67

```
SQLMAP and NOSQLMAP options are mutually exclusive. The
most recently specified option is in effect
```

SQL directives specifying the SQLMAP and NOSQLMAP options appear in the source code. The compiler applies the most recently specified option. If this is not your intent, specify only one of the two options.

68

```
Cannot access SSV
```

An incorrect ASSIGN SSV (Search SubVolume) number occurs. Specify the correct SSV number.

69 `Label declaration belongs inside a procedure or subprocedure`

A label declaration appears outside of a procedure or subprocedure. Move the label declaration into the appropriate procedure or subprocedure.

70 `Conflicting TARGET directive ignored`

A TARGET directive conflicts with a previous TARGET directive. The compiler accepts only the first TARGET directive that it encounters.

73 `An ASSIGN SSV number is too large`

A TACL ASSIGN command specifies an SSV value larger than 49. Use SSV values in the range 0 through 49; for example:

```
ASSIGN SSV48, \node1.$vol2.subvol3
```

74
```
The language attribute for this procedure conflicts with
a prior declaration
```

An EXTERNAL procedure declaration specifies language attributes that do not match those in the declaration that describes the procedure body.

75 `There are too many ASSIGN commands`

More than 75 ASSIGN commands appear in the compilation. Reduce the number of ASSIGN commands to 75 or fewer.

76 `Cannot use $OFFSET or $LEN until base structure is complete`

$LEN or $OFFSET is applied to an unfinished structure or to a substructure that is declared in an unfinished structure. The compiler does not calculate these values until the encompassing structure is complete. Complete the encompassing structure.

77 `SYMSERV died while processing the cross-reference listing`

The compiler has passed inconsistent symbol information to the Crossref process. Use the SYMBOLS directive for all your source code.

78

```
TAL cannot set RP value for forward branch
```

A CODE statement causes a conditional jump out of an optimized FOR loop. The compiler can generate correct code for the nonjump condition of the CODE statement, but not for the jump condition. To set the RP value for the forward branch, use a CODE (STRP ...) statement or an RP directive. .

79

```
Invalid directive option.  Remainder of line skipped.
```

An incorrect option appears in a directive. The compiler ignores the remainder of the directive line. Replace the incorrect option with a correct option.

80

```
This is a reserved toggle name
```

A reference to a reserved toggle name appears in the source code. Choose a different toggle name for your code.

81

```
The PAGES option of the SQL directive must be on the
command line
```

An SQL directive specifying the PAGES option appears in a directive line. The compiler ignores this directive. Specify this directive only in the compilation command.

83

```
Too many user-defined named toggles
```

You have exceeded the maximum number of named toggles. Use fewer named toggles.

84

```
Invalid parameter list
```

An incorrect parameter list appears in a procedure or subprocedure declaration. If your program is adversely affected, correct the parameter list.

86 `Return condition code must be 16-bit integer expression`

A procedure returns a condition code that does not evaluate to an INT expression. Correct the condition code so that it evaluates to an INT expression.

87 `The register stack should be empty, but is not`

The register stack should be empty after each statement (except CODE, STACK, and STORE). If you have not left items on the register stack deliberately, change your code.

88 `Address size mismatch`

This warning appears, for example, when:

- ☐ You pass the content of a standard (16-bit) pointer by reference to a procedure that expects the content of an extended (32-bit) pointer.
- ☐ You pass the content of an extended (32-bit) pointer by reference to a procedure that expects the content of a standard (16-bit) pointer.

89 `CROSSREF does not work with USEGLOBALS`

The CROSSREF and USEGLOBALS directives both appear in a compilation unit. The compiler issues warning 89 and turns off the CROSSREF directive. If you need to collect cross-reference information, remove the USEGLOBALS directive from the compilation unit.

90 `Initialization of UNSIGNED arrays is not supported`

An array of type UNSIGNED is initialized. Remove the initialization from the declaration of the UNSIGNED array.

91 `MAIN procedure cannot return a value`

A RETURN value appears in a RETURN statement in the MAIN procedure. When the MAIN procedure terminates, it calls a system procedure, so the return value is meaningless.

93

```
Do not use an SQL statement in a parametized DEFINE
```

An EXEC SQL statement appears in the text of a DEFINE that has parameters. The compiler evaluates the parameters and the SQL statement in the same buffer space.

94

```
A template structure is not addressable
```

The address of a template structure is specified. Specify only the address of a definition structure of a referral structure.

SYMSERV Messages

The following message might appear during compilation:

```
SYMSERV FATAL ERROR
```

This error appears only when the compiler detects a logic error within its operation. If you are using versions of the TAL compiler and SYMSERV from the same release, and if no other error message appears that would explain this behavior, please report this occurrence to Tandem. Include a copy of the complete compilation listing and the source, if possible.

BINSERV Messages

For BINSERV diagnostic messages, see the *Binder Manual*.

Common Run-Time Environment Messages

For Common Run-Time Environment (CRE) diagnostic messages, see the *CRE Programmer's Guide*.