

Pillar Language Specification

This document provides a complete description of the Pillar language.

**Digital Equipment Corporation - Confidential and Proprietary
For Internal Use Only**

Digital Equipment Corporation

**Digital Equipment Corporation—Confidential and Proprietary
For Internal Use Only**

November, 1988

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.


No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright ©1988 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem-10	PDP	VT
DECSYSTEM-20	PDT	
DECUS	RSTS	
DECwriter	RSX	

This document was prepared using VAX DOCUMENT, Version 1.1

Preface

Pillar is a high-level system programming language for use on 32-bit Digital Equipment Corporation systems, and for future 64-bit systems. Pillar was originally developed as part of the PRISM project, and then as part of the OSF project. Pillar was also a possible candidate for the DEC corporate implementation language.

Changes and scheduling constraints for the OSF project caused the design goals of Pillar to change slightly between the PRISM and OSF projects. During the PRISM project, an interim bootstrap language, called SIL, was developed to experiment with the concepts and design of the Pillar language. Pillar was to be quite different in syntax from the SIL language. Because of these recent constraints, however, Pillar is basically a superset of the SIL language.

Intended Audience

This specification is written to be a precise language specification of the Pillar language, and is intended for use by compiler writers and language experts to produce compilers and review the language. Many experienced programmers will be able to use this specification when writing code. However, it is not intended to be a user manual or a tutorial to learn the language.

Document Structure

This specification describes all of the features of the Pillar programming language. This specification contains the following chapters:

- Chapter 1 describes the notation used to define Pillar in the rest of the specification.
- Chapter 2 describes the lexical elements of Pillar, that is, the tokens out of which a Pillar program is constructed.
- Chapter 3 describes the naming and block structure rules of the language and introduces declarations.
- Chapter 4 describes the Pillar facilities that allow one to divide a program into several modules that share declarations.
- Chapter 5 describes the Pillar data types, their declaration, and representation in storage.
- Chapter 6 describes constants, literals, and the various types of constructors in Pillar.
- Chapter 7 describes values and variables in Pillar and how to declare them.
- Chapter 8 describes the storage allocation schemes in Pillar and how data is shared.
- Chapter 9 describes the various kinds of data references as they occur in Pillar programs.
- Chapter 10 describes Pillar expressions.
- Chapter 11 describes the different kinds of statements that can be written in Pillar.
- Chapter 12 describes Pillar blocks, their syntax and execution semantics, and the condition and exception facilities provided by Pillar.
- Chapter 13 describes Pillar procedures.

- Chapter 14 describes the target-specific features provided by Pillar.
- Appendix A contains a collection of the entire Pillar syntax.

CONTENTS

Preface	xiii
CHAPTER 1 NOTATION AND METHOD OF DEFINITION	1
1.1 Organization of the Definition	1
1.2 Errors, Exceptions, and Range Violations	2
1.3 Syntax Notation	2
1.3.1 Metasymbols in Syntax Definitions	2
1.4 Description of Operations	3
1.5 Variables and Symbols in Text	3
1.6 Remarks	4
CHAPTER 2 LEXICAL ELEMENTS	5
2.1 Character Set	5
2.2 Source Modules and Syntactic Analysis	8
2.3 Lexical Syntax	10
2.3.1 Identifiers	11
2.3.2 Numeric Literals	11
2.3.3 Character String Literals	12
2.3.4 Punctuation Symbols	13
2.4 Compile-Time Facility	13
CHAPTER 3 NAMING, BLOCK STRUCTURE, AND DECLARATIONS	15
3.1 Names	15
3.1.1 Reserved Names	16
3.2 Block Structure	16
3.3 Scope of a Declaration	17
3.3.1 Summary of Declarations and Scopes	17
3.4 General Declaration Principles	18
3.4.1 Factoring Declaration Keywords	18
3.4.2 Order of Declarations	19
3.4.3 Circular Declarations	19
3.4.4 Expressions in Declarations	19
CHAPTER 4 MODULES	21
4.1 Introduction to Pillar Modules	21
4.2 General Module Level Declarations	23
4.3 Importing Modules and Declarations	24

4.4 Program Modules	25
4.5 Definition Modules	25
4.5.1 Using Pillar Definition Modules in Other Languages	26
4.6 Implementation Modules	26
4.6.1 Declaration Completions	27
4.6.2 Implementation Without an Imported Declaration	27
4.7 External Declarations	27
4.7.1 Connecting the Declaration and the Completion	28
4.7.2 Connecting an external-opaque-type	28
4.7.3 Connecting external-procedures	28
4.8 Built-in Module	28
4.9 Module Options	29
4.9.1 Module Identification	29
4.9.2 Module Linkage Options	29
4.9.2.1 Default Conventions for Global Symbols	29
4.9.2.2 Qualified Globals Option	30
4.9.2.3 Global Synonym Option	30
4.9.2.4 Linker Value Option	31
4.9.3 Module Management	31
4.9.3.1 Module Consistency	32
CHAPTER 5 TYPES	35
5.1 Overview of Pillar types	35
5.1.1 Pillar's Type Structure	35
5.1.2 Named and Unnamed Types	36
5.2 Type Declarations	36
5.2.1 Type Specifications	37
5.3 Arithmetic Types	37
5.4 Ordinal Types	38
5.4.1 The Concept of Range	38
5.4.2 BOOLEAN	40
5.4.3 CHARACTER	40
5.4.4 The Types BYTE, WORD, LONGWORD and QUADWORD	40
5.4.5 Enumerated Types	40
5.4.6 Subrange Types	41
5.4.7 BIT	41
5.5 Set Types	42
5.6 Flexible Types	43
5.6.1 Bound Flexible Types	45
5.7 Pointer Types	46
5.8 String Types	46

5.9	Blank_DATA Types	47
5.10	Array Types	48
5.11	Record Types	49
5.11.1	Captured Extents	50
5.11.2	Unions and Variants	50
5.11.3	Unions	51
5.11.4	Variants	51
5.11.5	Record Extensions	52
5.12	STATUS	53
5.13	ANYTYPE	53
5.14	Procedure Types	53
5.15	Opaque Types	54
5.16	Relationships Among Types	55
5.16.1	Type Equivalence	55
5.16.2	Type Compatibility	56
5.16.3	Type Assignment Compatibility	56
5.16.4	Conversion Between Compatible types	56
5.16.5	Small Types and Constant Types	57
5.17	Data Representation	58
5.17.1	Standard Data Representation Rules	58
5.17.2	The SIZE Option	60
5.17.2.1	Rules for Size-options in Subrange Types	60
5.17.2.2	Rules for Size-options in Small Set Types	60
5.17.2.3	Rules for Size-options in Record Types	60
5.17.2.4	Rules for Size-options in Opaque Types	61
5.17.3	The LAYOUT Option	61
5.17.3.1	Determining Alignment Requirements	62
5.17.3.2	The Position Option	63
5.17.3.3	Filler Components	63
5.17.3.4	Variant Part and Union Layouts	64
CHAPTER 6 CONSTANTS, LITERALS, AND CONSTRUCTORS		65
6.1	Constant Declarations	65
6.2	Literal Constants	65
6.2.1	Literals with a Named Type	66
6.3	Initializers and Constructors	67
6.3.1	Initializers	67
6.3.1.1	NIL as an Initializer	67
6.3.2	Set Constructors	67
6.3.2.1	Set of Values	68
6.3.3	Array Constructors	68

6.3.4 Record Constructors	70
6.3.4.1 Using OTHERS in record-constructors	70
CHAPTER 7 VALUES AND VARIABLES	73
7.1 Overview of Values and Variables	73
7.2 Value Declarations	73
7.2.1 Value Completions	74
7.3 Variable Declarations	74
7.3.1 Variable Completions	75
7.4 BIND Declarations	75
7.5 DEFINE Declarations	76
CHAPTER 8 STORAGE ALLOCATION	79
8.1 Storage Classes	79
8.2 Data Sharing and Aliasing	80
8.3 Environments	80
8.3.1 Environment Declarations	80
8.3.2 Properties of Environments	81
8.3.3 Enabling an Environment	81
8.3.4 Procedures and Environments	81
CHAPTER 9 DATA REFERENCES	83
9.1 Syntax of Data References	83
9.2 Interpretation of References	83
9.2.1 Locations and Values	83
9.2.2 The Value of a Reference Rule	84
9.3 Reference to a Named Data Item	84
9.3.1 Reference to a Named Constant	84
9.3.2 Reference to an Element of an Enumerated Type	84
9.3.3 Reference to a Named Value	84
9.3.4 Reference to a Variable	84
9.3.5 Reference to a Bind Item	84
9.3.6 Reference to a Define Item	85
9.3.7 Reference to a Loop Control Variable	85
9.3.8 Reference to an IN Parameter	85
9.3.9 Reference to an IN Local Parameter	85
9.3.10 Reference to an OUT, IN OUT, BIND, or Result Local Parameter	85
9.3.11 Reference to a Procedure	85
9.3.12 Reference to a Condition	85
9.4 Reference to the Value of a Procedure Invocation	85
9.5 Reference to the Value of a Built-in Function Invocation	86

9.6 Indirect Reference	86
9.6.1 Dereferencing a Pointer Value	86
9.6.2 Implicit Dereferencing	86
9.7 Dot-qualified Reference	86
9.7.1 Reference to an Element of an Enumerated Type	87
9.7.2 Reference to an Extent of a Named Type	87
9.7.3 Reference to an Extent of a Parameter	87
9.7.4 Reference to the Length of a LIST Parameter or Local Parameter	87
9.7.5 Reference to an Element of an Environment	88
9.8 Indexed Reference	88
9.8.1 Reference to an Element of a LIST Local Parameter	89
9.9 Substring Reference	89
9.10 Type Cast Reference	89
9.11 Simple References	90
CHAPTER 10 EXPRESSIONS	91
10.1 Syntax of Expressions	91
10.2 Simple and Constant Expressions	93
10.2.1 Summary of Simple Expression Rules	93
10.2.2 Dynamic Values in Simple Expressions	93
10.2.3 Summary of Constant Expression Rules	94
10.3 Principles of Expression Evaluation	94
10.3.1 Order of Evaluation	94
10.3.2 Incomplete Evaluation	95
10.3.3 Evaluation of Integer Operations	95
10.3.4 Evaluation of Floating-Point Operations	95
10.4 Interpretation with a Target Type	95
10.5 Arithmetic Operations	96
10.5.1 Negation Operator	96
10.5.2 Addition Operator	96
10.5.3 Subtraction Operator	96
10.5.4 Multiplication Operator	96
10.5.5 Division Operator	96
10.5.6 Integer MOD Operator	97
10.5.7 Arithmetic Comparison Operators	97
10.5.8 Absolute Value Built-in Function	97
10.5.9 Integer Exponentiation Operator	97
10.5.10 SIGN Built-in Function	98
10.5.11 ODD Built-in Function	98
10.5.12 MAX Built-in Function	98
10.5.13 MIN Built-in Function	98
10.6 Boolean Operations	98

10.6.1 Boolean Complement Operator	98
10.6.2 AND Operator	98
10.6.3 OR Operator	99
10.6.4 Boolean Comparison Operators	99
10.6.5 Boolean Exclusive OR Operator	99
10.7 Ordinal Operations	99
10.7.1 Ordinal MAX Built-in Function	99
10.7.2 Ordinal MIN Built-in Function	99
10.7.3 Ordinal Comparison Operators	100
10.8 Set Operations	100
10.8.1 Set Complement Operator	100
10.8.2 Set Union Operator	100
10.8.3 Set Intersection Operator	100
10.8.4 Set Difference Operator	100
10.8.5 Set Exclusive OR Operator	101
10.8.6 Set Comparison Operators	101
10.9 Character String Operations	101
10.9.1 String Concatenation Operator	101
10.9.2 FIND_MEMBER Built-in Function	102
10.9.3 FIND_NONMEMBER Built-in Function	102
10.9.4 FIND_SUBSTRING Built-in Function	102
10.9.5 TRANSLATE_STRING Function	103
10.9.6 String Comparison Operators	104
10.10 Pointer Operations	104
10.10.1 Pointer Addition Operator	104
10.10.2 Pointer Subtraction Operator	104
10.10.3 ADDRESS Built-in Function	104
10.10.4 CONTAINING_RECORD Built-in Function	105
10.10.5 Pointer Comparison Operations	105
10.11 Operations Related to Types	106
10.11.1 MAX Built-in Function	106
10.11.2 MIN Built-in Function	106
10.11.3 DATA_TYPE_SIZE Built-in Function	106
10.11.4 FIELD_OFFSET Built-in Function	106
10.11.5 Conversion Functions	107
10.11.5.1 CONVERT_ORDINAL Built-in Function	107
10.11.5.2 CONVERT_POINTER Built-in Function	107
10.11.5.3 CONVERT_ARITHMETIC Built-in Function	108
10.11.5.4 CONVERT_SET Built-in Function	108
10.11.5.5 CONVERT_STRING Built-in Function	108
10.11.5.6 CONVERT_UNTYPED Built-in Function	109
10.11.6 INITIALIZE_FIELDS Built-in Function	109
10.12 Miscellaneous Built-in Functions	110
10.12.1 ZERO Built-in Function	110

10.12.2 ARGUMENT_PRESENT Built-in Function	110
10.12.3 VALIDATE_VALUE Built-in Function	110
10.12.4 VALIDATE_ALIGNMENT Built-in Function	111
CHAPTER 11 STATEMENTS	113
11.1 Control Flow and Statement Sequences	113
11.2 ASSERT Statement	114
11.3 ASSIGNMENT Statement	115
11.4 CASE Statement	115
11.5 Compound Statement	116
11.6 EXIT LOOP Statement	116
11.7 GOTO Statement	116
11.8 IF Statement	117
11.9 LOOP Statement	117
11.9.1 Loop Control by an Ordinal Type	118
11.9.2 Loop Control by Increment or Decrement	119
11.9.3 General Loop Control	120
11.10 NOTHING Statement	120
11.11 Built-in Function Call Statement	120
11.11.1 READ_REGISTER Built-in Function	121
11.11.2 WRITE_REGISTER Built-in Function	121
11.12 Procedure Call Statement	122
11.13 RAISE Statement	123
11.13.1 RAISE ERROR	123
11.13.2 RAISE EXCEPTION and RAISE REPORT	123
11.13.3 RAISE VECTOR	123
11.14 RETURN Statement	124
CHAPTER 12 LOCAL BLOCKS AND EXCEPTION HANDLING	125
12.1 Local Blocks	125
12.1.1 Declarations in a Local Block	125
12.1.2 Subprocedures	126
12.1.3 Termination of a Statement Sequence	126
12.1.3.1 Explicit Unwinding	126
12.1.3.2 Implicit Unwinding	126

12.1.4 Interpretation of a Local Block	126
12.1.4.1 The Enable Section of a Local Block	128
12.1.4.1.1 Acquiring a Lock	128
12.1.4.1.2 Disabling Alerts	128
12.1.4.1.3 Enabling and Disabling Underflow	128
12.1.4.1.4 Enabling a Condition Handler	129
12.1.4.1.5 Enabling a Message Vector	129
12.1.4.1.6 Enabling an Environment	129
12.2 Contents of the \$CONDITION Built-in Module	129
12.2.1 SEVERITY	130
12.2.2 STATUS	130
12.2.2.1 GET_SEVERITY Built-in Function	130
12.2.2.2 SUCCESS_STATUS Built-in Function	130
12.2.3 MESSAGE_VECTOR	131
12.2.3.1 GET_SEVERITY Built-in Function	131
12.2.3.2 GET_STATUS Built-in Function	131
12.2.3.3 GET_EXCEPTION_FLAG Built-in Function	132
12.2.3.4 SET_EXCEPTION_FLAG Built-in Function	132
12.2.3.5 SET_MESSAGE Built-in Function	132
12.2.3.6 EXPAND_MESSAGE_VECTOR Built-in Function	132
12.2.3.7 COPY_MESSAGE_VECTOR Built-in Function	133
12.2.4 CONDITION_HANDLER	133
12.2.5 CONDITION_DISPOSITION	133
12.2.6 USER_SEMAPHORE	133
12.3 Messages, Conditions, Exceptions, and Reports	134
12.3.1 Promotion of a Condition	134
12.3.2 Message Declarations	134
12.3.3 Condition Declarations	134
12.4 Procedural Condition Handling	135
12.5 Exception Handlers	136
12.6 Unwind Handlers	136
CHAPTER 13 PROCEDURES	137
13.1 Procedure Declarations	138
13.1.1 Procedure Type Specifications and Constructors	138
13.1.1.1 Parameter Repetitions	139
13.2 Parameters	139
13.2.1 IN Parameters	141
13.2.1.1 Special Restrictions on IN Parameters	142
13.2.2 OUT Parameters	142
13.2.3 IN OUT Parameters	142
13.2.3.1 STRING and VARYING_STRING, OUT and IN OUT Parameters	142
13.2.4 BIND Parameters	143

13.2.5 Matching Extents	143
13.2.5.1 The Normal Extent-Matching Rule	144
13.2.5.2 Extent Matching for CONFORM Arrays	145
13.2.6 Parameters with Captured Extents	146
13.2.7 Parameter Options	146
13.2.8 The CONFORM Option	146
13.2.9 Parameter Default Values	147
13.2.10 OPTIONAL Parameters	148
13.2.11 LIST Parameters	148
13.2.11.1 Argument Interpretation for LIST Parameters	149
13.2.12 KEYWORD Parameters	149
13.2.13 STATUS VECTOR Parameters	150
13.3 Procedure Results	151
13.3.1 Procedures Returning STATUS	151
13.4 Procedure Invocations	152
13.4.1 Argument Lists	152
13.4.2 Argument List Validation	154
13.5 INLINE and INLINE ONLY Procedures	155
13.6 EXTERNAL Procedures	156
13.6.1 Procedure Completions	156
13.7 Environments and Procedures	157
13.8 Procedure Linkages	157
13.8.1 Argument Passing Mechanisms and Pillar Conventions	157
13.8.2 Linkage Specifications	158
CHAPTER 14 TARGET-SPECIFIC FEATURES	161
APPENDIX A COLLECTED SYNTAX	163
GLOSSARY	
INDEX	
TABLES	
2-1 Pillar Character Set	5
2-2 Pillar Keywords	9
2-3 Pillar Punctuation Symbols	13
3-1 Reserved Names	16
5-1 Primitive Types	35
6-1 Default Types for Literal-Constants	66
6-2 Initial DEFAULT Field Values	71

CHAPTER 1

NOTATION AND METHOD OF DEFINITION

1.1 Organization of the Definition

For the most part, the definition of Pillar (in this manual) is divided into units, each unit describing a particular language construction. The description begins with a template showing the form of the construction. The template is either a syntactic category definition or an operation template. It is followed by text explaining the construction's interpretation, and this generally refers to the interpretation of subphrases named in the template. For example, the definition of an assignment-statement begins:

■ assignment-statement

data-reference = expression ;

The data-reference is interpreted to yield a location that must be *assignable*. The location has a type *t*, which is used as the target type while evaluating the expression.

In this example, the interpretation of each subphrase is qualified in some way; the data-reference is interpreted to obtain an *assignable location*, and the expression is interpreted with a *target type* (see Section 5.16.2). In connection with expressions and statements, the terms *evaluation* and *execution* are also used to mean *interpretation*.

For the purposes of definition, the interpretation of a Pillar source module can be divided into two steps:

1. The module is analyzed as an instance of the module syntactic category. If not rejected for being an error, the module can now be treated as a tree structure without regard to details of concrete representation, such as punctuation. This step is primarily defined by the syntax; however, lexical analysis and the compile-time facilities add some nonanalytic elements to the process. This is covered in Chapter 2.
2. The module is dynamically interpreted in an environment containing storage, other modules, external communication devices, and such. As explained above, the rules for this interpretation are tied to the syntax.

When a module is compiled, the compiler carries out Step 1 and as much as is possible of Step 2. It is not practical to specify exactly what the compiler does, but certain things are guaranteed, especially regarding error detection and the evaluation of constant-expressions.

Technically, the syntax is ambiguous. However, the semantic rules that apply at compile time eliminate the ambiguities.

1.2 Errors, Exceptions, and Range Violations

An *error* is a violation of a Pillar language rule. Most errors are detected by the compiler. Errors that might not be detected by the compiler are classified as exceptions, range violations, or errors with unpredictable consequences.

An *exception* happens during interpretation of a language construction; it abruptly suspends and then terminates the interpretation. Continuing from the point of exception is not allowed. Recovery is possible using condition handling features that apply at the local-block level.

A *range violation* is an error that will cause an exception if the module containing the range violation is compiled with range checking enabled. If range checking is not enabled, the error has unpredictable consequences. As far as possible, the compiler checks for range violations at compile time, whether or not range checking is enabled. Except within a declaration, such detection does not generally prevent successful compilation, because the offending part of the module might never be executed.

1.3 Syntax Notation

Pillar has a two-level syntax. The lexical syntax defines the division of a line of text into tokens and white-space, the latter being ignored after lexical analysis. The terminal symbols of the lexical syntax are individual characters.

The main syntax of Pillar defines the structure of a Pillar source module. The terminal symbols of this syntax are the tokens produced by lexical analysis.

Both levels of syntax use the same notation. A few syntactic categories are defined informally, the remainder by definitions of this form:

■ category-name
 constructive-definition

Within these definitions, category-names are set in lowercase and a sans-serif typeface, with compound names hyphenated. Specific symbols are in uppercase; these symbols never contain blanks.

1.3.1 Metasymbols In Syntax Definitions

A few symbols in the language are similar to symbols in the syntax notation. In the syntax notation, such language symbols are delimited by quotation marks (which are not language characters); for example: '}'. To avoid ambiguity, symbols are also delimited this way if two specific language symbols occur next to each other in a syntax definition. In addition to the apostrophe, the following symbols are used as metasymbols in syntactic definitions.

Metasymbol	Meaning
{ }	Braces denote a set of alternatives, one of which is used. The alternatives are displayed on separate lines within the braces. (It is possible to have only one alternative within the braces; this is useful with the ellipsis metasymbols.)

Metasymbol	Meaning
[]	Double brackets denote a set of choices, none or one of which is used. The alternatives are displayed on separate lines within the brackets.
...	Ellipses are used after a closing brace or bracket to denote possible repetition: one or more occurrences after a brace, zero or more occurrences after a bracket. When the braces or brackets enclose more than one alternative, each occurrence can be a different alternative.
...	The occurrences are not separated by any additional symbols.
,...	The occurrences are separated by commas.
;...	The occurrences are separated by semicolons.

1.4 Description of Operations

In many cases, an operator or built-in function is generic in the sense that it accepts a variety of data types; its meaning for one data type can be quite different from that for another. Typically, the description of the operator or function's meaning for a particular data type is given in a section with other operations on the same data type. Thus, the description of "+" as an arithmetic operation is given in the section on arithmetic operations; "+" as the string concatenation operator is described in the section on string operations.

The description of an operator or built-in function begins with a display showing a template invocation of the operation. Text stating which data types are applicable to this case of the operator or function will follow the description, or will be presented at the beginning of a section if the data types are the same for an entire group; for example, the set difference operator is defined this way:

$$\text{result} = x - y ;$$

The result contains every element that is a member of x but not a member of y .

At the beginning of the Set Operators section is this paragraph:

The operands of set operators must have set types with the same range. (Requiring that the set ranges be identical eliminates situations that produce very complex code; the CONVERT_SET function [see Section 10.11.5.4] can be used to explicitly adjust ranges.) Unless specified otherwise, the result's type is a set type with the same range as the operands.

1.5 Variables and Symbols in Text

Some parts of the manual use an informal mathematical style of exposition in explaining the language rules. Names that are variables of this exposition are set in italics; for example:

If the target type t ...

Here, the variable name t denotes a type occurring in the interpretation of an arbitrary Pillar program.

In the text, normal mathematical symbols have their conventional mathematical meaning; they are not Pillar operators; for example:

When $m/3$...

If m has the value 4, $m/3$ has the value one-and-one-third (which cannot be represented exactly by any standard Pillar data type).

Built-in Pillar objects (types, constants, functions, and such) and Pillar keywords are denoted by their standard Pillar names in uppercase. Thus, "INTEGER" is the name of a built-in Pillar type, while "integer" is just an English word.

When a fragment of a Pillar program appears in this manual, it is set between paragraphs, unless it is very small, in which case it is enclosed in quotation marks, as in:

If the expression "X[I] + 2"...

1.6 Remarks

Throughout this language specification are remarks from the language designers. These remarks are printed in italics and surrounded by triple backslashes, for example: `\\\ Here is what a remark looks like. \\\`

CHAPTER 2

LEXICAL ELEMENTS

This chapter describes Pillar's character set, lexical syntax, and compile-time facility.

2.1 Character Set

Pillar is based on an 8-bit character set.

Pillar lists all the characters with character codes 0 through 127. Those characters listed as nonescaped characters can be used directly in character-string-literals, or for other purposes, as shown in the lexical syntax. In addition, the quotation mark ("), back slash (\), horizontal tab (H_T), and form feed (F_F) characters are significant in the syntax; no other characters are allowed in Pillar source modules. All 256 characters are allowed in character string *data*, and there is a convenient escape notation for them in character-string-literals.

Table 2-1: Pillar Character Set

Terminal Graphic	Decimal Value	Nonescaped Character?	Escape Name	Description
	0		NUL	Null character
	1		SOH	Start of heading
	2		STX	Start of text
	3		ETX	End of text
	4		EOT	End of transmission
	5		ENQ	Enquiry
	6		ACK	Acknowledge
	7		BEL	Bell
	8		BS	Backspace
H_T	9		HT	Horizontal tab
L_F	10		LF	Line feed
V_T	11		VT	Vertical tab
F_F	12		FF	Form feed
C_R	13		CR	Carriage return
	14		SO	Shift out
	15		SI	Shift in
	16		DLE	Data link escape
	17		DC1	Device control 1

Table 2-1 (Cont.): Pillar Character Set

Terminal Graphic	Decimal Value	Nonescaped Character?	Escape Name	Description
	18		DC2	Device control 2
	19		DC3	Device control 3
	20		DC4	Device control 4
	21		NAK	Negative acknowledge
	22		SYN	Synchronous
	23		ETB	End of transmission
	24		CAN	Cancel
	25		EM	End of medium
	26		SUB	Substitute
	27		ESC	Escape
	28		FS	File Separator
	29		GS	Group separator
	30		RS	Record separator
	31		US	Unit separator
	32	<i>yes</i>		Space
!	33	<i>yes</i>		Exclamation point
"	34			Quotation mark
#	35	<i>yes</i>		Number sign
\$	36	<i>yes</i>		Dollar sign
%	37	<i>yes</i>		Percent sign
&	38	<i>yes</i>		Ampersand
'	39	<i>yes</i>		Apostrophe
(40	<i>yes</i>		Opening parenthesis
)	41	<i>yes</i>		Closing parenthesis
*	42	<i>yes</i>		Asterisk
+	43	<i>yes</i>		Plus sign
,	44	<i>yes</i>		Comma
-	45	<i>yes</i>		Hyphen and minus sign
.	46	<i>yes</i>		Period and decimal point
/	47	<i>yes</i>		Slash
0	48	<i>yes</i>		Zero
1	49	<i>yes</i>		One
2	50	<i>yes</i>		Two
3	51	<i>yes</i>		Three
4	52	<i>yes</i>		Four
5	53	<i>yes</i>		Five
6	54	<i>yes</i>		Six
7	55	<i>yes</i>		Seven
8	56	<i>yes</i>		Eight
9	57	<i>yes</i>		Nine
:	58	<i>yes</i>		Colon

Table 2-1 (Cont.): Pillar Character Set

Terminal Graphic	Decimal Value	Nonescaped Character?	Escape Name	Description
;	59	yes		Semicolon
<	60	yes		Less than
=	61	yes		Equal sign
>	62	yes		Greater than
?	63	yes		Question mark
@	64	yes		At sign
A	65	yes		Uppercase A
B	66	yes		Uppercase B
C	67	yes		Uppercase C
D	68	yes		Uppercase D
E	69	yes		Uppercase E
F	70	yes		Uppercase F
G	71	yes		Uppercase G
H	72	yes		Uppercase H
I	73	yes		Uppercase I
J	74	yes		Uppercase J
K	75	yes		Uppercase K
L	76	yes		Uppercase L
M	77	yes		Uppercase M
N	78	yes		Uppercase N
O	79	yes		Uppercase O
P	80	yes		Uppercase P
Q	81	yes		Uppercase Q
R	82	yes		Uppercase R
S	83	yes		Uppercase S
T	84	yes		Uppercase T
U	85	yes		Uppercase U
V	86	yes		Uppercase V
W	87	yes		Uppercase W
X	88	yes		Uppercase X
Y	89	yes		Uppercase Y
Z	90	yes		Uppercase Z
[91	yes		Opening bracket
\	92			Back slash
]	93	yes		Closing bracket
^	94	yes		Circumflex
_	95	yes		Underscore
`	96	yes		Grave accent
a	97	yes		Lowercase a
b	98	yes		Lowercase b
c	99	yes		Lowercase c

Table 2-1 (Cont.): Pillar Character Set

Terminal Graphic	Decimal Value	Nonescaped Character?	Escape Name	Description
d	100	yes		Lowercase d
e	101	yes		Lowercase e
f	102	yes		Lowercase f
g	103	yes		Lowercase g
h	104	yes		Lowercase h
i	105	yes		Lowercase i
j	106	yes		Lowercase j
k	107	yes		Lowercase k
l	108	yes		Lowercase l
m	109	yes		Lowercase m
n	110	yes		Lowercase n
o	111	yes		Lowercase o
p	112	yes		Lowercase p
q	113	yes		Lowercase q
r	114	yes		Lowercase r
s	115	yes		Lowercase s
t	116	yes		Lowercase t
u	117	yes		Lowercase u
v	118	yes		Lowercase v
w	119	yes		Lowercase w
x	120	yes		Lowercase x
y	121	yes		Lowercase y
z	122	yes		Lowercase z
{	123	yes		Opening brace
	124	yes		Vertical line
}	125	yes		Closing brace
~	126	yes		Tilde
DEL	127		DEL	Delete, rubout

2.2 Source Modules and Syntactic Analysis

A Pillar *source module* is a sequence of lines, each line a sequence of characters. The module is interpreted as an occurrence of the module syntactic category in the following way:

- Each line must satisfy the lexical syntax for a line; this determines the tokens within the line.
- The tokens in all lines form a single token sequence. (This eliminates the division into lines and eliminates any white-space that occurred between tokens.) From this point on, each token is treated as a terminal symbol.
- If the token sequence contains any compile-time-facility symbols, the token sequence is transformed according to Pillar's compile-time-facility rules. The result of this step must satisfy the syntax for the module category. Pillar's compile-time-facility is very primitive, and is described in Section 2.4.

8 Lexical Elements

The compiler actually interleaves the preceding steps, but the effect is the same as if they were sequential.

A token is any literal, identifier, or punctuation symbol (all these categories are defined in the lexical syntax). The various categories of literals are used directly in the main syntax. Punctuation symbols are used as specific terminal symbols.

Identifiers are used two ways in the main syntax: as names, or as specific terminal symbols (as listed in Table 2-2). Note that some keywords are *reserved words*; they cannot in general be used as names (that is, they are not allowed as an instance of the *name* category).

Table 2-2: Pillar Keywords

Keywords in **boldface** are *not* reserved in the Pillar compiler.

ALIAS ED	ENVIRONMENT	MOD	RETURN
ALIGNED	ERROR	MODULE	RETURNS
ALIGNMENT	EXCEPTION	NEXT	REVEAL
AND	EXIT	NIL	SET
ANYTYPE	EXTENDS	NOT	SHARED
ARGUMENT	EXTERNAL	NOTHING	SIZE
ARRAY	EXTENTS	OF	STATUS
ASSERT	EXTENSIBLE	ONLY	STRING
BEGIN	FATAL	OPAQUE	SUBPROCEDURES
BIND	FILLER	OPTIONAL	SUCCESS
BIT	FOR	OPTIONS	THEN
BIT_DATA	GLOBALS	OR	TO
BOOLEAN	GOTO	ORDER	TRAILING
BY	HANDLER	OTHERS	TRUNCATE
BYTE	IDENTIFICATION	OUT	TYPE
BYTE_DATA	IF	OVERLAY	UNDERFLOW
CAPTURE	IMPLEMENT	PACKED	UNION
CASE	IMPORT	PILLAR	UNWINDING
COMPONENTS	IN	PILLAR\$_ASSERT	VALUE
CONDITION	INFORMATIONAL	PILLAR\$_ERROR	VARIABLE
CONFORM	INLINE	POINTER	VARIANTS
CONSTANT	INTEGER	POSITION	VARYING_STRING
DEFAULT	KEYWORD	PROBING	VECTOR
DEFINE	LARGE_INTEGER	PROCEDURE	WARNING
DESCRIPTOR	LAYOUT	PROGRAM	WHEN
DISABLE	LINKAGE	QUADWORD	WHILE
DOUBLE	LINKER	QUADWORD_DATA	WITH
DOWN	LIST	QUALIFIED	WORD
ELSE	LOCK	RAISE	WORD_DATA
ELSEIF	LONGWORD	REAL	XOR
ENABLE	LONGWORD_DATA	RECORD	

Table 2-2 (Cont.): Pillar Keywords

END	LOOP	REFERENCE
ENTRY	MESSAGE	REPORT

2.3 Lexical Syntax

Pillar's lexical syntax defines how a line is partitioned into tokens. A line can contain a form feed (at the beginning only), tokens, white-space (spaces and horizontal tabs) and a comment (at the end only). White-space can appear before, between, and after tokens.

Note that the representation of lines in the operating system environment is not covered by the Pillar language definition. Troublesome characters, such as line feed, are excluded from Pillar source modules so that they need not be handled by lexical analysis.

■ line

[F_F] [white-space] [token [white-space]]... [comment]

■ token

{
 identifier
 compile-time-facility-symbol
 decimal-literal
 binary-literal
 octal-literal
 hexadecimal-literal
 floating-point-literal
 character-string-literal
 punctuation-symbol
}

Nonpunctuation tokens must be separated by white-space or punctuation symbols; they must not be adjacent. For example,

```
FOR i=1 TO 10 LOOP
```

is legal, whereas

```
FORi = 1TO10LOOP
```

is not.

■ white-space

{
 space
 tab
}...

A comment begins with an exclamation point. The rest of the line can contain any of the characters allowed in a source module except a form feed.

■ comment

$$! \left[\begin{array}{l} \text{non-escaped-character} \\ \text{tab} \\ \backslash \\ " \end{array} \right] \dots$$

A non-escaped-character denotes a character with a character code in the range 32–126 or 160–255, excluding the quotation mark (") and back slash (\). (Non-escaped-characters are indicated in Table 2–1.) The tab category stands for a single horizontal tab character.

2.3.1 Identifiers

Identifiers are composed of letters, decimal digits, underscores (_), and dollar signs (\$). The first character cannot be a digit.

■ identifier

$$\left\{ \begin{array}{l} \text{letter} \\ \$ \\ _ \end{array} \right\} \left[\begin{array}{l} \text{letter} \\ \$ \\ _ \\ \text{decimal-digit} \end{array} \right] \dots$$

An instance of the letter category is one of the characters A–Z or a–z. Uppercase and lowercase are equivalent in identifiers; “a” is replaced by “A,” “b” by “B,” and so forth. An instance of the decimal-digit category is one of the digits 0–9.

2.3.2 Numeric Literals

The data type assigned to a numeric literal depends on the literal’s form and the context in which it occurs; the rules are in Section 6.2.1 and Chapter 10. In numeric literals, the digits 0–9 are used with their normal meaning. The letters A–F and a–f are used as hexadecimal digits.

An instance of the categories decimal-, binary-, octal-, or hexadecimal-digit-string denotes a sequence of one or more of the indicated class of digits. Such a digit string denotes a nonnegative integer value according to the normal conventions for numbers in its base.

■ decimal-literal

decimal-digit-string

■ binary-literal

$$" \{ \text{binary-digit-string} \} \dots " \left\{ \begin{array}{l} \text{b} \\ \text{B} \end{array} \right\}$$

■ octal-literal

$$" \{ \text{octal-digit-string} \} \dots " \left\{ \begin{array}{l} \text{o} \\ \text{O} \end{array} \right\}$$

■ hexadecimal-literal

" { hexadecimal-digit-string }... " { X
X }

A floating-point-literal denotes a rational number, and it uses the conventional decimal floating-point notation. The exact value can be changed when converting to the internal form, which depends on the type associated with the literal.

■ floating-point-literal

{ decimal-digit-string exponent
decimal-digit-string . decimal-digit-string [exponent]
.decimal-digit-string [exponent] }

■ exponent

{ e } [+] decimal-digit-string
{ E } [-] decimal-digit-string

2.3.3 Character String Literals

■ character-string-literal

" [non-escaped-character
escaped-character]... "

The value of a character-string-literal is the sequence of characters between the quotation marks, with each occurrence of an escaped-character replaced by the character it denotes. The data type depends on the number of characters and the context in which the literal occurs; the rules are in Section 6.2.1 and Chapter 10.

■ escaped-character

{ \\
\
\(decimal-digit-string)
\(identifier) }

The first two cases denote the backslash and quotation mark, respectively. The third case denotes the character whose character code is the indicated integer, which must be in the range 0..255. In the final case, the identifier (mapped to uppercase) must be one of the escape names listed in Pillar, and it denotes the corresponding character. For example,

"ABC\ "DEF\ \GHI\ (126)\ (HT) "

denotes the string

ABC"DEF\GHI~H
T

See Table 2-1 for a list of escape names.

2.3.4 Punctuation Symbols

An instance of the punctuation-symbol category is any of the sequences of characters shown in Table 2-3.

Table 2-3: Pillar Punctuation Symbols

()	*	+	-
/	.	,	;	:
{	}	[]	^
=	>	<	**	..
::	<=	=>	<>	==
(+)				

2.4 Compile-Time Facility

The only compile-time facility in the Pillar compiler is a `%INCLUDE` feature; its form is:

```
%INCLUDE character-string-literal
```

The string must be a file specification, using the host system rules, optionally followed by `/LIST` or `/NOLIST` to include or not include the file in the compiler listing; for example:

```
%INCLUDE "somefile"/LIST
```

On a VMS system this includes the file named "SOMEFILE" from the current default directory. The default file type for an included file is `.PILLAR`.¹

¹ Pillar has a complete compile time facility already designed. However, due to schedule constraints, it will not be released until after V1 Pillar.

CHAPTER 3

NAMING, BLOCK STRUCTURE, AND DECLARATIONS

3.1 Names

In the Pillar syntax, the *name* category occurs at all the points where an identifier is to be interpreted as a name in accordance with Pillar's scope rules. In general, a reserved name, for example INTEGER, or a qualified name, for example *alpha.x*, can be used at the same points, so the name category covers all possibilities.

■ name

$$\left\{ \begin{array}{l} \text{unqualified-name} \\ \text{qualified-name} \\ \text{reserved-name} \end{array} \right\}$$

■ unqualified-name

identifier

An occurrence of *unqualified-name* must be within the scope of a declaration of the identifier; the *unqualified-name* denotes the innermost such declaration. However, if the *unqualified-name* is within the scope of more than one declaration, and the one with innermost scope is the local declaration of a procedure parameter, the occurrence of *unqualified-name* is an error; *qualified-name* must be used instead. For an explanation of the Pillar principles governing multiple declarations of the same identifier, see Section 3.3.

■ qualified-name

name . identifier

There are three cases:

1. The name denotes an imported definition module. That module must export a declaration of the identifier. This qualified-name denotes that exported declaration.
2. The name denotes the current module. The identifier must be declared at module level in the current module. This qualified-name denotes that declaration.
3. The name denotes a procedure, and this qualified-name occurs within the procedure's body. The identifier must be declared as one of the procedure's parameters. This qualified-name denotes the local parameter declaration of that parameter.

Cases 2 and 3 are provided only to deal with situations where, unavoidably, a procedure parameter has the same name as a higher-level declaration.

Note that the dot notation is used more generally in the category data-reference.

3.1.1 Reserved Names

■ reserved-name

identifier

Here identifier is one of the reserved names listed in Table 3-1. As names, these reserved words always denote the built-in declarations; they cannot be redeclared.

Table 3-1: Reserved Names

Reserved words that can be used as names.

ANYTYPE	DOUBLE	PILLAR\$_ERROR	VARYING_STRING
BIT	INTEGER	QUADWORD	WORD
BIT_DATA	LARGE_INTEGER	QUADWORD_DATA	WORD_DATA
BOOLEAN	LONGWORD	REAL	
BYTE	LONGWORD_DATA	STATUS	
BYTE_DATA	PILLAR\$_ASSERT	STRING	

3.2 Block Structure

A *block* is a module, procedure-body, or compound-statement. A declaration that is within a block, *b*, is at *block-level* in *b* provided that it is not contained within another declaration in *b* or within a subblock of *b*. Two other cases produce block-level declarations:

- The declarations of elements within an enumerated-type-constructor are block level declarations unless the QUALIFIED option is used.
- If *p* is a procedure with a parameter *X*, a local parameter declaration corresponding to *p* is implicitly created in *p*'s procedure-body.

\\\ *To some extent, the distinction between a parameter's declaration in a procedure-type-constructor, and the corresponding local declaration in the procedure-body is a technicality of the method used in the language specification. *

The following example shows some declarations and their positions in the block structure.

```
MODULE alpha;
TYPE t : RECORD           ! t declared at block level in alpha
  x : integer;           ! x not declared at block level
END RECORD;
TYPE color : (           ! color declared at block level in alpha
  red);                 ! red declared at block level in alpha
  switch : (
  red) QUALIFIED;       ! This red not declared at block level

PROCEDURE p(             ! p declared at block level in alpha
  IN x : integer);      ! x not declared at block level here
                       ! Implicit declaration of x as a local parameter

VARIABLE y : integer;   ! y declared at block level in the body of p
BEGIN
  ...                   ! Some of p's code
  WITH                 ! Start a compound-statement with declarations
  VARIABLE z : integer; ! declared at block level in the compound-statement
  BEGIN
  ...                   ! Compound-statement's code
  END;                 ! End of the compound-statement
```

```
END PROCEDURE p;  
END MODULE alpha;
```

A block is the scope for its block-level declarations (see Section 3.3). The block also affects the allocation of storage for variables and the rules for expressions within block-level declarations. The key distinction here is between modules and the other kinds of blocks. A declaration that is at block-level in a module is a *module-level declaration*; a declaration at block-level in a procedure-body or compound-statement is a *local declaration*.

Module-level declarations are completely interpreted at compile time. Expressions within them must, in general, be constant. Module-level variable-declarations have static storage.

In general, a local declaration cannot be completely interpreted until the containing procedure-body or compound-statement is executed. An expression within a local declaration can denote a value that is determined dynamically by evaluating the expression during the block's prologue. Storage for a local variable declaration is allocated only for the duration of the block's execution. (This is a bit over simplified; for more details see Chapter 8).

3.3 Scope of a Declaration

Each declaration in a module has a *scope*. This is the lexical interval in which the declared identifier can be used as an unqualified-name. The language rules are such that scopes nest: if two scopes overlap, they are the same or one properly includes the other. The rules governing multiple declarations with the same name are as follows.

- There is never a name conflict between declarations with disjoint scopes.
- It is an error to have two declarations with the same name and the same scope.
- A *nested declaration* is one whose scope is properly included in the scope of another declaration of the same identifier. Nested block-level declarations are not allowed except for the local parameter declarations created implicitly in a procedure body.
- Nested declarations of labels or loop control variables are not allowed.

The nested declaration rule enhances the readability of code. If an unqualified-name in code is within the scope of a declaration, it must refer to that declaration; one does not have to worry about overlooking a nested declaration of the same name. The exception allowing nested declarations of local procedure parameters does not invalidate this principle, because of the special rule (see Section 3.1) that requires the use of a qualified-name within the scope of such a nested declaration. This exception is made for procedure parameters, because their names are determined by the procedure type, and it might not be feasible to change the names (in the parameter list) to avoid conflicts.

3.3.1 Summary of Declarations and Scopes

- The current module's name is declared with the entire module as the declaration's scope.
- The scope of a module-level declaration is the entire current module.
- Importing a module, either in an import-section or implement-section, declares the imported module and all explicitly named components. The scope of these declarations is the entire current module.

- Declarations exported by a module imported into the current module do not have a scope in the current module unless they are explicitly imported using the *COMPONENTS* construction. However, such a declaration can be referred to using a *qualified-name*.
- The scope of a procedure parameter declaration (see Section 13.2) is the procedure type constructor containing the declaration. In addition, a local parameter declaration is created implicitly in the body of each procedure of that procedure type (see Section 13.1). The scope of this local declaration is the procedure body.
- The scope of a block-level declaration in a local-block (procedure body or compound statement) is the entire local-block.
- The scope of a loop control variable (see Section 11.9) is the entire loop statement containing it.
- The scope of a label (see Section 11.1) is the statement sequence in which it occurs, unless it occurs in the main statement sequence of a procedure body. In the latter case, the scope is the entire procedure body.
- An enumerated type constructor declares one or more enumerated elements (see Section 5.4.5). If *QUALIFIED* is not used in the enumerated type constructor, the scope of these declarations is the entire block (local-block or module) containing the enumerated type constructor. If *QUALIFIED* is used, the scope is the enumerated type constructor.
- A flexible type declaration declares one or more extent parameters of the type. (see Section 5.6). The scope of such an extent parameter declaration is the part of the flexible type declaration following the colon after the flexible type's name (which is lexically in the containing block scope).
- The scope of a record field declaration or filler declaration (see Section 5.11 and Section 5.17.3.3) is the entire record type constructor in which it is declared. Outside of the type constructor, fields are referred to using the dot notation in a data-reference.
- The scope of a member variable declaration is the environment declaration that contains it. Member variables are referred to using the dot notation in a data-reference.

3.4 General Declaration Principles

3.4.1 Factoring Declaration Keywords

In Pillar, block-level declarations begin with a distinctive keyword, for example, *CONSTANT* or *PROCEDURE*. When two or more declarations of the same kind are adjacent, only the first requires the keyword, for example:

```
VARIABLE
  x : integer;
  y : POINTER t;
PROCEDURE
  p(...) EXTERNAL:
  q(...) INLINE EXTERNAL;
```

3.4.2 Order of Declarations

In some scopes, the order of declarations is significant, for example in a parameter list. However, there is no general requirement that a declaration occur after all other declarations on which it depends. For example, the following declarations order is allowed.

```
VARIABLE x : string(count);  
CONSTANT count = 10;  
PROCEDURE p ( IN s : string(n); IN n : integer[0..]);
```

3.4.3 Circular Declarations

Circular declarations are disallowed; a declaration cannot depend on itself. Dependence is defined recursively: one declaration depends on another if the first depends directly on the second, or if it depends directly on a third declaration which in turn depends on the second. If a declaration, *D1*, contains a name that denotes a declaration, *D2*, then *D1* depends directly on *D2*, with one exception. The exception is this: use of a pointer-type-reference does not cause dependency on the associated type of the referred to pointer type unless that is the same declaration. Examples:

```
TYPE  
r: POINTER r;           ! r is a circular declaration  
v: RECORD               ! v is circular (through w.k, below)  
  i: integer;  
  j: w;  
END RECORD;  
w: RECORD               ! w is circular (through v.j, above)  
  k: v;  
END RECORD;  
s: RECORD               ! s is not circular  
  f: POINTER s;  
  i: integer;  
END RECORD;  
u(e: integer): RECORD  ! u is not circular  
  f: POINTER u(2*e);  
  i: integer;  
END RECORD;  
t(e: integer): RECORD  ! t is not circular  
  CAPTURE e;  
  f: POINTER t;  
  i: integer;  
END RECORD;
```

3.4.4 Expressions in Declarations

Pillar allows expressions in declarations, but such an expression must be *simple* according to rules explicitly stated in Section 10.2. The rules have been chosen so that a simple expression can easily be computed at compile time when the operands are constants, and so that this class of expressions is sufficient for effective use of Pillar's flexible type facility. The rules eliminate explicit side effects (which can result from use of function procedures), and they minimize the chance of an exception during the expression's evaluation.

Expressions within module-level declarations are generally required to be constant expressions; they are evaluated at compile time. The two exceptions are an expression depending on the extent parameter of a flexible type, and an expression within a define-declaration; in both cases, evaluation of the expression is not required until the declaration is referred to.

CHAPTER 4

MODULES

4.1 Introduction to Pillar Modules

A module is the normal compilation unit in Pillar. A source module is one text file. Where necessary, the term *current module* is used to distinguish the module being compiled from other modules.

\\\ A source module does not have to be a conventional text file. For example, it could be stored in a data base, or it could be transiently generated within the compiler by use of a compile-time facility. In due course, the compiler will support compiling a set of Pillar modules to obtain a single object module. These and other variations do not affect the structural partitioning of a program or system into a set of modules. \\\

There are three forms of modules.

■ module

{ program-module
definition-module
implementation-module }

A program module is used for the main module of a conventional program. Definition modules and implementation modules are of much greater interest in system programming.

A Pillar definition module *exports* declarations for use by other modules. These declarations tie together the set of modules making up a complete program or system. Compiling a Pillar definition module produces an *information module*, which contains the exported declarations in compiled form.

To use an exported declaration called *ed*, from a definition module called *alpha*, in a second module, the second module's import section (Section 4.3) names alpha as an *imported* module. The import section can also specify *ed* explicitly as an imported component, in which case it can be referred to by its unqualified name. Otherwise, the qualified name notation must be used, for example, *alpha.ed*.

As with other languages, compiling a source module can produce an object module containing the code and data required in any program or system that uses the module. An object module will be produced for a definition module if it contains a complete module-level value-, variable-, message-, condition-, or procedure- declaration. Such definition modules are handy during program development and in casual programming, but they are generally not used in production quality systems.

Note that compiling a definition module can produce an object module containing information for the debug symbol table.

Pure definition modules can be obtained by using external-declarations. Such a declaration provides type information, but it does not give a complete description of the declared object. The remainder of the description is given in a separate implementation module, or outside of Pillar. The category name for the remainder of the description ends in "-completion," for example, "procedure-completion."

The following examples illustrate these variations. In the first one, the object module *random_numbers* will contain the code for the procedure *uniform* and an object language definition of the global symbol for the variable *random_seed*.

```
MODULE random_numbers; ! Will produce an object module
VARIABLE random_seed : longword; ! Exported static variable
PROCEDURE uniform( ) RETURNS double; ! Exported procedure
BEGIN
    ! code to compute a new value for the variable random_seed and return
    ! this value as a 64-bit floating point value in the range (0..1)
END PROCEDURE uniform;
END MODULE random_numbers;
```

In the next example, the procedure and static variable are declared by external declarations. Therefore, there will be no code or data in the object module *random_numbers*:

```
MODULE random_numbers; ! Will normally not produce an object module.
VARIABLE random_seed : longword EXTERNAL; ! Exported static variable
PROCEDURE uniform( ) RETURNS double EXTERNAL; ! Exported procedure
END MODULE random_numbers;
```

In the next example, two external-procedures are used. There will be no code or data in the object module *random_numbers*:

```
MODULE random_numbers; ! Will normally not produce an object module.
PROCEDURE uniform( ) RETURNS double INLINE EXTERNAL; ! External procedure
PROCEDURE set_seed(IN new_seed : longword) EXTERNAL; ! External procedure
END MODULE random_numbers;
```

If the random number sequence can be assumed as properly initialized, any one of the three versions of the module *random_numbers* can be imported in the following example:

```
PROGRAM xxx ENTRY main;
IMPORT random_numbers COMPONENTS uniform;
PROCEDURE main();
...
BEGIN
    ...
    x = 2*uniform();
    ...
END PROCEDURE main;
END MODULE xxx;
```

The version of the module *random_numbers* with *uniform* and *random_seed* declared external can be implemented as follows:

```
MODULE random_impl;
IMPLEMENT random_numbers COMPONENTS random_seed, uniform;

VARIABLE random_seed = 1; ! implementation of the declaration of random_seed
                           ! this allocates static storage and initializes it

PROCEDURE uniform() RETURNS; ! header introducing the implementation of uniform
BEGIN
    ! code to compute a new value for the variable random_seed and return
    ! this value as a 64-bit floating point value in the range (0..1)
END PROCEDURE uniform;

END MODULE random_impl;
```

The version of the module *random_numbers* with *uniform* as an external inline procedure can be implemented as follows:

```
MODULE random_impl;
IMPLEMENT random_numbers COMPONENTS uniform, set_seed;

VARIABLE seed : longword; ! private static variable

PROCEDURE uniform() RETURNS;
BEGIN
    ! code to compute a new value for the variable seed and return
    ! this value as a 64-bit floating point value in the range (0..1)
END PROCEDURE uniform;

PROCEDURE set_seed(new_seed);
BEGIN
    seed = new_seed;
END PROCEDURE set_seed;

END MODULE random_impl;
```

To compile a module that uses the inline procedure *uniform*, the compiler (automatically) loads the information module for *random_impl*. The object module from this compilation must be linked against the object module for *random_impl*, because it defines the storage for the private static variable *seed*. The code for *set_seed*, which was declared external, is also in the *random_impl* object module.

4.2 General Module Level Declarations

These declarations are allowed at module level (see Section 3.2) in any form of module.

■ general-module-level-declaration

```
{
  constant-declaration
  normal-type-declaration
  flexible-type-declaration
  procedure-type-declaration
  complete-value-declaration
  complete-variable-declaration
  environment-declaration
  define-declaration
  complete-message-declaration
  complete-condition-declaration
  complete-procedure-declaration
  external-declaration
}
```

4.3 Importing Modules and Declarations

Any form of module can contain an import section.

■ import-section

```
[ IMPORT { module-import } ... ] ...  
[ REVEAL { name } ,... ; ]
```

Each module-import (if any) imports modules into the current module. In addition, individual declarations can be explicitly imported by using a component list.

The names following REVEAL (if present) must denote opaque types. These opaque types are *revealed* in the current module (see Section 5.15). There are two restrictions:

- If the current module contains the declaration of the type as an external opaque type, revealing it is not allowed (because the module that implements the type depends on the current module).
- If the current module is an implementation module that implements an external opaque type, that type cannot be (redundantly) revealed in the import section.

■ module-import

```
identifier [ COMPONENTS component-list ] ;
```

The identifier is declared as a module imported into the current module. The identifier cannot be given any other declaration at module level, and giving another module-import for the same module is not allowed. However, the current module's implement-section can import the same module (hence declare the same identifier). In this case, the component-list must be present in this module-import (which otherwise would be redundant).

The component-list names one or more declarations exported from the module imported by this module-import. Each of these declarations is *explicitly imported* into the current module. In the current compilation, the explicitly imported declaration's scope is the entire current module.

Some components from the imported module can be imported by the current module's implement section. If so, explicitly naming such a component in this component-list is not allowed (because it would be redundant). However, a wild-card match *is* allowed, but a wildcard component must match some declaration that was not explicitly imported in the implement-section.

Note that an exported declaration called *x*, from an imported module called *alpha*, can be referred to by its qualified name, *alpha.x*, regardless of whether *x* is explicitly imported as a component of *alpha*.

■ component-list

```
{ identifier [ * ] },...
```

A component-list specifies a set of one or more exported declarations within an imported module, *M*. Each identifier must occur only once in the component-list and must be the name of an exported declaration in *M*.

The construction *identifier** is used for wildcard importing; it denotes all exported declarations whose names begin with the characters in the identifier. There must be at least one such declaration.

Note that the identifier can be a reserved word, for example,

```
COMPONENTS fab$, FOR*;
```

selects all declarations whose names begin with "fab\$" or "for".

Wildcard importing is generally used with definition modules exporting names that are governed by prefix conventions.

4.4 Program Modules

■ program-module

```
PROGRAM identifier ENTRY name [ module-identification ] ;  
[import-section ]  
{ general-module-level-declaration }...  
[linkage-option-section]  
END PROGRAM identifier;
```

The identifier following PROGRAM is declared as the name of the current module. The same identifier must occur after the END PROGRAM that terminates the program module.

A program module is used for the main module of a conventional program. It neither exports declarations nor implements declarations from other modules.

The name following ENTRY must denote a complete procedure declaration (at module level in the current module). The name's type must satisfy the target-system requirements for main program entries.

4.5 Definition Modules

■ definition-module

```
MODULE identifier [ module-identification ] ;  
[import-section ]  
{  
  general-module-level-declaration  
  complete-opaque-type-declaration  
  external-declaration  
}...  
[linkage-option-section]  
END MODULE identifier;
```

The identifier following MODULE is declared as the name of the current module. The same identifier must occur after the END MODULE that terminates the definition module.

A definition module *exports* declarations for use in other modules. All the module-level declarations in the current module are exported. No other declarations are exported.

Note that declarations imported (from another module) into the current definition module are not reexported.

4.5.1 Using Pillar Definition Modules in Other Languages

Pillar is a OZIX/OSF replacement for SDL, a language used for common definition files on VMS. There will be a definition module utility (derived from the Pillar compiler) that reads Pillar information modules and translates them into other OZIX/OSF supported languages.

*\\\ If worthwhile, a few extras might be added to Pillar just to clarify the translation without changing the Pillar meaning. At least some of the OZIX/OSF compilers will directly accept the information modules. This might also sharpen the translation. *

4.6 Implementation Modules

■ implementation-module

```
MODULE identifier [ module-identification ] ;  
{ implement-section }...  
[import-section ]  
{ general-module-level-declaration  
  declaration-completion }...  
[linkage-option-section]  
END MODULE identifier;
```

The identifier following MODULE is declared as the name of the current module. The same identifier must occur after the END MODULE that terminates the implementation module.

This form of module does not export any declarations. It implements external-declarations from definition-modules specified in the implement-section. The keyword IMPLEMENT at the beginning of the implement-section distinguishes this form of module from a definition-module.

■ implement-section

```
{ IMPLEMENT { module-implement } ... } ...
```

Each module-implement in the implement-section designates a module to be imported into the current module, and designates one or more exported declarations within it that are to be implemented (completed) in the current module.

■ module-implement

```
identifier COMPONENTS component-list ;
```

The identifier is declared as a module imported into the current module. The identifier cannot be given any other declaration at module level, and giving another module-implement for the same module is not allowed. However, it can be specified again in the current module's import-section.

The component-list names one or more exported declarations within the imported module that are to be implemented in the current module. If a component in the list is a particular identifier, the corresponding declaration must be external. In the case of a wildcard component, *word **, any matching declarations that are not external are ignored, but at least one external declaration must be matched.

*\\\ Using a wildcard here appears to be a controversial style question. *

The declarations named by the component list are explicitly imported into the current module. The current module must contain a completion for each of these declarations. It is invalid to explicitly import any of them again in the current module's import section.

Opaque types imported in an implement section are revealed in the current module.

4.6.1 Declaration Completions

■ declaration-completion

{ opaque-type-completion
value-completion
variable-completion
procedure-completion }

Declaration completions occur only in an implementation module. They are described in the chapters that cover the corresponding forms of declarations.

4.6.2 Implementation Without an Imported Declaration

A special interpretation is given to the identifier *\$nodefinition* when used as the module name in a module-implement. It means that the identifiers specified in the component-list do not have exported declarations in a Pillar definition module. For each of these identifiers, the implementation module must contain a complete module-level value-, variable-, or procedure- declaration. The identifier is defined to the linker as an unqualified global symbol denoting the declaration's data or procedure entry. This feature is provided so that certain run-time library routines and complex code sequences can be implemented in Pillar without creating exported declarations that should never be used.

Wildcards cannot be used with *\$nodefinition*; there is nothing to match. This identifier is treated as the name of a built-in module; it cannot be used as the name of any other module.

4.7 External Declarations

■ external-declaration

{ external-value-declaration
external-variable-declaration
external-message-declaration
external-condition-declaration
external-procedure-declaration
external-opaque-type-declaration }

An external-declaration is allowed only at module level. In general, the declaration only provides type information. For example, an external-procedure-declaration gives the procedure's type (parameter declarations, result type, and others) but not the the procedure's body. The implementation of an external-declaration is normally given in a Pillar implementation module. This module provides the completion of the external declaration. Code and data for the completion is in the object module produced by compiling the implementation module.

The implementation of an external value-, variable-, or procedure- declaration can be made outside Pillar by linking the associated global symbol to an appropriate data item or routine.

The item must be consistent with the Pillar declaration and with Pillar implementation conventions.

One important rule pertains to completing external variable declarations: *If an external variable declaration is linked outside Pillar, the data to which it is linked must not overlap the data to which another exported name is linked.*

The specific forms of external declarations are described in Chapter 7, Section 13.6, and Section 12.3.

4.7.1 Connecting the Declaration and the Completion

The following external declarations are always connected to their completion by the linker.

- External-value-declaration
- External-variable-declaration
- External-message-declaration
- External-condition-declaration

The linker makes the connection between an object module using an external declaration and the object module containing its implementation. The compiler does not access the implementation of an external-declaration that is used in the current module.

4.7.2 Connecting an external-opaque-type

The compiler does not need the implementing information module for an external-opaque-type-declaration unless the type is revealed in the current module.

4.7.3 Connecting external-procedures

If an external-procedure-declaration is an inline-procedure, the compiler requires access to the information module created by the Pillar implementation module, otherwise the external-procedure-declaration will be connected to its body by the linker.

The module implementing an external-opaque-type-declaration or an `INLINE` external-procedure-declaration must import (through the `implement-section`) the definition module containing the declaration. Therefore, the definition module cannot itself use these two external declarations except in the case of using an unrevealed opaque type.

4.8 Built-in Module

Pillar has one built-in module, `$CONDITION`, which contains built-in declarations related to condition handling.

This built-in module must be imported in an `import-section`. Its name cannot be used as a name for conventional modules. The module `$CONDITION` is described in Section 12.2.

In the text of this language specification, items in these modules are generally denoted by the qualified name, for example, "`$Pillar.initialize_fields`". In code examples, the unqualified names are generally used, because this is the normal coding practice.

4.9 Module Options

This section describes miscellaneous options that can be used in modules. The syntax definitions for the three forms of modules show where the options occur.

4.9.1 Module Identification

■ module-identification

IDENTIFICATION (character-string-literal)

The string is used as the identification string in the current module's object module and information module. The string must satisfy the system requirements (target, host, or both) for such identification strings. If this option is not present, the default string "NO-IDENT" is used.

4.9.2 Module Linkage Options

■ linkage-option-section

LINKAGE OPTIONS
[qualified-globals-option]
[global-synonym-option]
[linker-value-option]

This section occurs just before the final END in a module. If present (that is, if the keywords LINKAGE OPTIONS occur), then at least one of the options must be present.

These options modify the normal Pillar conventions for the use of global symbols. These are the symbols used in the target system's object language for references between object modules.

4.9.2.1 Default Conventions for Global Symbols

The circumstances in which Pillar uses a global symbol, and the default conventions for that symbol, are as follows.

- An exported value- or variable- declaration: The global symbol's name is the same as the declaration's name (unqualified). The global symbol denotes the address of the data.
- An exported noninline procedure declaration: The global symbol's name is the same as the declaration's name (unqualified). The global symbol denotes the procedure entry in accordance with the target system's conventions.
- An exported message- or condition- declaration: \The rules here are TBS.\
- An external declaration in a program module or implementation module: This is treated as an exported declaration of the same kind.
- The entry symbol for a program module: The global symbol's name is the same as the declaration's name (unqualified).
- A value-, variable-, message-, condition-, or noninline-procedure- declaration needed for expansion of an external inline procedure: These are nonexternal declarations in a module, *M*, implementing the external inline procedure. With one exception, these declarations are at module level in *M*; the exception is a value declaration within the procedure whose size exceeds 64 bits. The global symbols for these private declarations

have qualified names of the form *M.qual.dcl_name*, where *dcl_name* is the declaration's unqualified name and *qual* is an appropriately unique name provided by the Pillar compiler.

- The module-level declarations for identifiers specified as components in an IMPLEMENT \$nodefinition: The default convention for the global symbol is the same as for an exported declaration of the same kind.

*\\\ The OZIX/OSF object language supports qualified names for global symbols. On VAX this is an open issue. *

4.9.2.2 Qualified Globals Option

■ qualified-globals-option

QUALIFIED GLOBALS;

This option specifies that certain global symbols should have names qualified by the current module's name.

In a definition module, the option applies to all declarations exported from the module.

In an implementation module, the option applies to module-level declarations for identifiers specified as components in an IMPLEMENT \$nodefinition. If there are no such components, the option is an error.

The option is not allowed in a program module.

*\\\ If this option is not used, one can encounter name conflicts when linking even though there are no name conflicts as far as Pillar is concerned. However if the names exported from the module are all governed by a prefix convention, for example, all begin with "my_facility\$", no conflicts should occur during linking. The use of name prefixes is so well established in DEC software engineering, that this governs the default naming convention for global symbols. *

4.9.2.3 Global Synonym Option

■ global-synonym-option

{name = character-string-literal} ,... ;

This option specifies an explicit name for a global symbol. Each name in the option must be the name of an appropriate module-level declaration, and the corresponding character-string-literal must be a valid global symbol name in accordance with target system conventions.

In a definition module, this option can be applied to any exported value-, variable-, message-, condition-, or noninline-procedure- declaration.

In an implementation module, this option can be applied to an external declaration or to a module-level declaration for an identifier specified as a component in an IMPLEMENT \$nodefinition.

In a program module, this option can be applied to an external declaration.

The global-synonym-option overrides all other options in determining the global symbol's name.

4.9.2.4 Linker Value Option

■ linker-value-option

```
LINKER VALUE ( { name } ,... ) ;
```

Each name must denote an appropriate module-level declaration. In a definition module, it must be an exported value declaration; in an implementation or program module it must be an external value declaration. In either case, the value's type must be constant with size ≤ 32 bits.

This option specifies that the value declaration's global symbol is to denote the actual value rather than the address of the value. In the object language the value is, in effect, defined as a 32-bit value. If the declaration's type is an integer type, the signed 32-bit representation is used. For other types, the value is zero-extended to 32 bits as required.

In general, use of this option is not recommended on OZIX/OSF.

4.9.3 Module Management

When a non-built-in module is imported, the compiler must find and load its information module. The rules for this are part of the compiler's command interface specification, and they might depend on the host system. The imported module must be compiled before the current module, which rules out a pair such as:

```
MODULE alpha;
IMPORT beta;
TYPE t : RECORD x : POINTER beta.s; END RECORD;
END MODULE alpha;

MODULE beta;
IMPORT alpha;
TYPE s : RECORD
  y : POINTER alpha.t;
END RECORD;
END MODULE beta;
```

Information modules can be given explicitly to the compiler as input files or the compiler can find them by library search. The set of explicit input modules is treated as the first library in the library search. The module name is the key in the search, so it is not possible to import two modules with the same name.

\\\ The compiler is designed so that it can be extended to support other methods, for example, obtaining the module from a data base. The language can be extended to handle duplicate module names if that proves necessary, for example:

```
"IMPORT alpha = duplicate-module_name"\  
\\\
```

If an imported definition module depends on a second definition module, the compiler might need to import the second module and modules on which it depends. Consider the compilation of a module, *M*, that imports and uses the procedure *p* from the module *alpha*.

```
MODULE alpha;
IMPORT beta;
PROCEDURE p( IN x : beta.t) INLINE EXTERNAL;
END MODULE alpha;
```

Because *M* uses the procedure *p*, the compiler needs to get the type declaration of *t* from the module *beta*. The compiler automatically finds and loads the information module for *beta* using the same search rules it would use for an explicit import of *beta*. The name *beta* is the key in this search.

Because *p* is an external inline procedure, the compiler also needs to get the body of *p*, which is in some implementation module, *IMP*. The name of *IMP* is not specified in the definition module *alpha*. The compiler finds the information module for *IMP* by library search using the qualified name *alpha.p* as the key in a library index reserved for this purpose.

The same library set is used in all these searches.

4.9.3.1 Module Consistency

Modules are typically modified and recompiled in the course of program development, and this can lead to inconsistencies. For example, suppose that a program module, *p*, uses definition modules *alpha* and *beta*, and suppose that *alpha* also uses *beta*. Consider the following sequence of events:

1. *Beta* is compiled.
2. *Alpha* is compiled.
3. *Beta* is changed and recompiled.
4. *p* is compiled.
5. The program *p* is linked.

Because *alpha* was not recompiled, the information modules for *alpha* and *beta* can be inconsistent when *p* is compiled, or the corresponding object modules can be inconsistent when the program is linked, which can cause serious problems. In particular, an undetected inconsistency at compile time can cause the generation of faulty code, although it is more likely to simply cause the compiler to abort mysteriously.

Production programs and systems should be built from scratch using methods that make the occurrence of such inconsistencies impossible, but doing this during development every time something is changed is too cumbersome. For this reason the compiler checks the consistency of the modules used in a compilation. It also has an option to simply check the consistency of a set of modules with each other and with all the modules on which they depend.

The consistency checking is based on computing a *signature* for each exported declaration. If two declarations have the same signature, then, with very high probability, they are equivalent for all practical purposes. If the signatures are different, the declarations are certainly not equivalent.

\\\ The current plan is to use 64-bit signatures. \\\

The signatures of a definition module's exported declarations are stored in its information module. When a module, *M*, uses an exported declaration, that usage, including the signature, is stored in *M*'s information module and object module. Consistency checking is just a matter of comparing signatures. It does not require knowledge of the rule for computing signatures.

Library searches for the implementations of external declarations can lead to odd results. Suppose an old module, *alpha*, implements *X* and *Y*, while a newer module, *beta*, implements *Y*. Assume the search rules are set up to search a library of new modules first. If the compiler, first searches for *Y* and then for *X*, it will load module *beta* and then module *alpha*. While loading *alpha*, it will detect the duplicate implementation of *Y*. However, if the first search is for *X*, the compiler loads *alpha*, which also implements *Y*, so it will not bother with the formal library search for *Y* unless this degree of consistency checking is requested.

The compiler will have options to control the degree of consistency checking, including an option to check a specified set of modules and all others on which they depend, directly or indirectly.

CHAPTER 5

TYPES

This chapter describes Pillar's type structure and type declarations.

5.1 Overview of Pillar types

A type is a property of a data item. It is a set of values that can be assumed by the item, together with a set of operations that can be performed on the item. For example, `INTEGER` and `REAL` are types.

Each type also has associated with it a size and arrangement of its data in storage. These properties are referred to as its *data representation*.

5.1.1 Pillar's Type Structure

Pillar provides a set of primitive types, and also allows the construction of new types. Each primitive type is denoted by a reserved word. These primitive types are summarized in Table 5-1. Some of the types in this table have a parameter; these types are flexible types, which are described in Section 5.6.

Table 5-1: Primitive Types

Primitive Type	Values of that Type
<code>ANYTYPE</code>	A datum of no specific size or type
<code>BIT_DATA(<i>n</i>)</code>	A datum consisting of <i>n</i> bits
<code>BOOLEAN</code>	A boolean value
<code>BYTE</code>	A single byte containing an unsigned 8-bit integer
<code>BYTE_DATA(<i>n</i>)</code>	A datum consisting of <i>n</i> bytes
<code>CHARACTER</code>	A single byte containing an 8-bit character
<code>DOUBLE</code>	A 64-bit floating-point number
<code>INTEGER</code>	A 32-bit signed integer
<code>LARGE_INTEGER</code>	A 64-bit signed integer
<code>LONGWORD</code>	A single longword containing an unsigned 32-bit integer
<code>LONGWORD_DATA(<i>n</i>)</code>	A datum consisting of <i>n</i> longwords
<code>QUADWORD</code>	A single quadword containing an unsigned 64-bit integer
<code>QUADWORD_DATA(<i>n</i>)</code>	A datum consisting of <i>n</i> quadwords

Table 5-1 (Cont.): Primitive Types

Primitive Type	Values of that Type
REAL	A 32-bit floating-point number
STATUS	A datum containing a system status value
STRING(<i>n</i>)	A string of <i>n</i> characters
WORD	A single word containing an unsigned 16-bit integer
WORD_DATA(<i>n</i>)	A datum consisting of <i>n</i> words
VARYING_STRING(<i>n</i>)	A string of characters whose length is $\leq n$

In addition to providing primitive types, Pillar allows one to define new types by using the various kinds of type constructors defined in this chapter.

5.1.2 Named and Unnamed Types

A Pillar type can be named or unnamed. In its most simple form, a *named type* is either a primitive type or a type declared in a type declaration. The concept *named type*, however, refers to more than this, and a complete definition of it is given in Section 5.2.1. A named type can be used in the type specification of other declarations.

In contrast, an unnamed type is introduced by a type constructor in a declaration other than a type declaration. For example, consider the following declarations, the syntax of which is covered later in this chapter:

```
TYPE
    positive_integer: integer [1..];
VARIABLE
    positive_var: positive_integer;
    negative_var: integer [..-1];
```

Here, the type declaration introduces a named-type, *positive_integer*. The variable *positive_var* is declared with this named type. However, the variable *negative_var* has an unnamed type.

5.2 Type Declarations

A type declaration that occurs as one of the categories normal-type-declaration, complete-opaque-type-declaration, external-opaque-type-declaration, or procedure-type-declaration declares an identifier as a new named type. Pillar also has flexible-type-declarations, which declare an identifier as the name of a family of flexible types.

A normal-type-declaration declares an identifier as a new named type that is not opaque (opaque types are described in Section 5.15).

■ normal-type-declaration

```
[ TYPE ] identifier :
{
    type-specification
    enumerated-type-constructor
};
```

The keyword TYPE can be omitted only if this type declaration immediately follows another type declaration.

5.2.1 Type Specifications

A type-specification is a named type or one of several kinds of type constructors. Type specifications are generally used to specify a new type in declarations other than type declarations. Type specifications (or an enumerated type constructor) also can be used to specify the type in a normal type declaration.

■ type-specification

$$\left\{ \begin{array}{l} \text{named-type} \\ \text{subrange-type-constructor} \\ \text{set-type-constructor} \\ \text{array-type-constructor} \\ \text{record-type-constructor} \end{array} \right\}$$

A named-type (see Section 5.1.2) is one of the following:

- A primitive type
- A type declared in a normal or opaque type declaration
- A bound-flexible-type that is either a binding of a primitive flexible type, or a type declared in a flexible-type-declaration (bindings of flexible types are discussed in Section 5.6.1)
- POINTER *t*, where *t* is one of the cases above

Syntactically, a named-type is described as follows:

■ named-type

$$\left\{ \begin{array}{l} \text{[POINTER] name} \\ \text{bound-flexible-type} \end{array} \right\}$$

If name occurs, it must denote a type *t*, where *t* is:

- A type that is not a flexible type
- A flexible type with captured extents, unless:
 - this is an occurrence of a named-type used as the type-specification of a VALUE or VARIABLE declaration or
 - this is an occurrence of a named-type immediately following EXTENDS in a record-type-constructor (see Section 5.11.5).

In either of these cases, however, *name* can be supplied as *name* in the bound-flexible-type (see Section 5.6.1).

If name occurs, the named-type denotes the type *t* or POINTER *t*.

A bound-flexible-type denotes a particular instance of a flexible type (see Section 5.6).

5.3 Arithmetic Types

The primitive arithmetic types provided by Pillar are INTEGER, LARGE_INTEGER, REAL, and DOUBLE. These types, together with any type derived from them, are *arithmetic types*. Pillar provides a standard set of arithmetic operations that can be applied to data items of an arithmetic type.

Integer types, one subcategory of arithmetic types, are one of the primitive types `INTEGER` or `LARGE_INTEGER`, or any type derived from them. The values taken by an integer type are the integers in some interval.

`LARGE_INTEGER` denotes signed 64-bit integers, and `INTEGER` denotes signed 32-bit integers. The integer types are also ordinal types. Subrange-type-constructors (see Section 5.4.6) allow the construction of a new type derived from another integer type, and enable specifying precise limits for an integer type's range.

Floating-point types, the other subcategory of arithmetic types, are one of the types `REAL` or `DOUBLE`. The values taken by a floating-point type are rational numbers. `REAL` denotes 32-bit floating point numbers, and `DOUBLE` denotes 64-bit floating point numbers.

5.4 Ordinal Types

An ordinal type is a type in which each possible value of the type is one of a contiguous range of integers. Pillar provides the following ordinal types:

- Base ordinal types:
 - The primitive types `INTEGER` and `LARGE_INTEGER`. These are also integer types.
 - The primitive types `BOOLEAN`, `CHARACTER`, `BYTE`, `WORD`, `LONGWORD` and `QUADWORD`. None of these are integer types.
 - Enumerated types. An enumerated-type-constructor introduces a new distinct base ordinal type.
- Subrange types. A subrange-type-constructor introduces a new ordinal type that is derived from another ordinal type given in the subrange-type-constructor.

Fundamental to ordinal types is the concept of *base ordinal type*. Base ordinal types are only those types listed as such in the above list.

Every ordinal type t has a base ordinal type. If the ordinal type t is a base ordinal type, then its base ordinal type is t . If the ordinal type t is a subrange type, then its base ordinal type is the base ordinal type of the ordinal type given in the subrange-type-constructor.

An ordinal type takes on values in an interval determined by the type: $low \leq value \leq high$. However, except for the integer types, ordinal values are not allowed in arithmetic operations because they are used for conceptually different purposes; this is reflected in the compatibility rules for ordinal types (defined in Section 5.16.2). The low and high values of a named ordinal type can be obtained using the `MIN` and `MAX` built-in functions.

The phrase "ordinal value of" is used to refer to the natural integer value of an ordinal item. The ordinal value of v is also referred to as `ORD(v)`.

5.4.1 The Concept of Range

The concept of a *range* of ordinal values is fundamental to dealing with ordinal types. A range can be regarded as a pair of ordinal values (low, high) that have the same base ordinal types and that satisfy:

$$ORD(low) \leq ORD(high) + 1$$

The base ordinal type of the low and high ordinal values is called the range's type. A range can also be regarded as the set of all values x having the range's type and satisfying:

$$low \leq x \leq high$$

The number of elements in the set above is:

$$ORD(high) - ORD(low) - 1$$

There are several contexts in Pillar where the range-specification category is used to specify a particular range. The notation allows one to specify the low and high values, the low value and the number of elements, or the name of an ordinal type, which indicates the same range as that type. In the first case, the low and high values can be taken by default from a target type.

■ range-specification

$$\left\{ \begin{array}{l} \text{[expression] .. [expression]} \\ \text{expression : (expression)} \\ \text{named-type} \end{array} \right\}$$

There are three classes of range-specifications.

Class 1: The range-specification contains “..”

The expressions (if present) must have ordinal types with equivalent base ordinal types. If the context provides a target type, its base ordinal type must be equivalent to the expressions' base ordinal types. The value of the first expression becomes the low value in the range, and the value of the second expression becomes the high value in the range. If either expression is omitted, the context must provide a target type, and the corresponding limit value is that of the target type's range. The low and high values must satisfy:

$$ORD(low) \leq ORD(high) + 1$$

Otherwise, a range violation occurs. Note that if equality holds in the above relationship, the range contains no elements, and is said to be an *empty range*. For example, an empty range can be specified like this:

[2..1]

but never like this:

[3..1]

Class 2: The range-specification contains “:”

The first expression must have an ordinal type, and this must be compatible with the target type (if the context provides a target type). The value of the first expression is the low value in the range. The second expression must have an integer type; its value is the number of elements in the range. If the second expression is negative, a range violation occurs. If the second expression is zero, the range is empty.

Class 3: The range-specification is a type-name

The type-name must be declared as the name of an ordinal type. The range is the same as that ordinal type's range.

All Classes

In all of the preceding classes, if there is a target type, the range's low and high values must lie within the target type's range (otherwise, a range violation occurs).

In some places that use a range, a *constant range* is required. A constant range is one whose specification contains only constant expressions.

5.4.2 BOOLEAN

The primitive ordinal type **BOOLEAN** takes on values representing true and false in logical operations. These values have the predeclared names **TRUE** and **FALSE**; their ordinal values are:

```
ORD (FALSE) = 0
ORD (TRUE)  = 1
```

Boolean values can also be represented by the type **BIT** (described in Section 5.4.7).

5.4.3 CHARACTER

The primitive ordinal type **CHARACTER** takes on values in the Pillar character set. The ordinal value of a **CHARACTER** value is its character code.

5.4.4 The Types **BYTE**, **WORD**, **LONGWORD** and **QUADWORD**

The primitive ordinal types **BYTE**, **WORD**, **LONGWORD**, and **QUADWORD** correspond to fundamental units of storage. These types are used when no particular type interpretation is appropriate for the data item (see Section 5.9). Their ordinal values lie in the following ranges:

```
BYTE      0 .. 28 - 1
WORD      0 .. 216 - 1
LONGWORD  0 .. 232 - 1
QUADWORD  0 .. 264 - 1
```

5.4.5 Enumerated Types

An enumerated type is an ordinal type defined by an enumeration of elements. The type is ordinal because each element is associated with a unique integer value.

A named enumerated type is introduced by a type-declaration containing an enumerated-type-constuctor; this is the only context in which such a constructor occurs (Pillar does not have unnamed enumerated types).

■ enumerated-type-constuctor

```
( { identifier } , ... ) [ QUALIFIED ]
```

Let the identifiers in the constructor be e_1, e_2, \dots, e_n . The new enumerated type has n distinct values. The identifier e_k is declared as a named constant (see Section 6.1) denoting the k th value in the type:

```
ORD (ek) = k - 1
```

e_k is also called an *element* of the enumerated type. ¹

If the keyword **QUALIFIED** is present in an enumerated-type-constructor, then the scope of the declaration of e_k is the type constructor itself. An element of such a type must be referred to from within the type's scope by the qualified name $t.e_k$.

If the keyword **QUALIFIED** is not present, then the scope of the declaration of e_k is the same as the scope of the containing type-declaration. In this case, using a qualified name to refer to one of the type's elements from within the type's scope is permitted, but not necessary.

Qualified names have another use in referring enumerated types. If the enumerated type t is imported into module m , then an element of t , e_k that is not imported into module m can be referred to from that module by the qualified name $t.e_k$.

5.4.6 Subrange Types

A subrange-type-constructor introduces a new ordinal type whose base ordinal type is an existing type. The new type's range of values is given by an explicit range-specification.

■ subrange-type-constructor

```
{ name [range-specification] [ size-option ] }  
{ name size-option }
```

In the form that contains the range-specification, the name must denote an ordinal type. This type is used as the target type for interpretation of the range-specification, and this type's base ordinal type becomes the new subrange type's base ordinal type. The range-specification, which supplies the range of the new subrange type, must be a constant range-specification, and not a named-type. ²

The size-option, if present, controls the internal representation of the subrange type (see Section 5.17.2). If present, the subrange type must be one that would have been a small type (see Section 5.16.5) if the size-option were not present.

In the form of subrange-type-constructor without the range specification, the name must denote a small (see Section 5.16.5) subrange type. The size-option controls the internal representation of the new subrange type (see Section 5.17.2).

5.4.7 BIT

BIT is a reserved word that denotes a special predeclared type. This type takes on boolean values, occupies one bit of storage, and is an ordinal type. The type **BIT** is declared as if the following declaration were present (this declaration is not legal in Pillar because **BIT** is a reserved word):

```
TYPE  
  bit: boolean size(bit);
```

¹ Like all ordinal types, the values that an enumerated type can assume are contiguous. A possible future extension of the language is to define a variation of enumerated types, in which the elements do not necessarily have contiguous values.

² There does not seem to be any reason to allow subrange types with nonconstant range-specifications, and doing so would complicate ordinal types. Given this, is there any reason to allow an empty range-specification here?

A data item of type BIT is not addressable and is called a *bit-class type*. The allowed usage of such data types is very restricted; essentially, they are allowed only as the types of array elements and record fields. BIT_DATA is also a bit-class type, as are subranges and small set types specified with a bit size.

5.5 Set Types

A set type represents a set of ordinal values. It is characterized by a range. The set type's base ordinal type is the base ordinal type of the range. A set type's values are sets of ordinal values whose elements lie in the set type's range. The ordinal values of the set's range must all lie within the range of the type INTEGER (otherwise, a range violation occurs).

A set-type-constructor introduces a new set type.

■ set-type-constructor

```
{ SET name [range-specification] [ size-option ] }  
{ name size-option }
```

For example:

```
TYPE  
  int_set: SET integer [0..63];  
  bool_set: SET boolean [false..true];
```

In the form that contains the range-specification, the name must denote an ordinal type. This type is used as the target type for interpretation of the range-specification, and this type's base ordinal type becomes the new set's base ordinal type. The range-specification supplies the range of the new set type. The ordinal values in the range must all lie within the range of the type INTEGER (otherwise, a range violation occurs).

The range-specification need not be constant, and must not be a named-type.

The size-option, if present, controls the internal representation of the set type (see Section 5.17.2). If it is present, the set type must be one that would have been a small type (see Section 5.16.5) if the size-option were not present.

In the form of set-type-constructor without the range specification, the name must denote a small set type. The size-option controls the internal representation of the new set type (see Section 5.17.2).

Pillar supports basic operations on set types (see Section 10.8). For example, if x and y are set variables,

```
x = x + y;
```

assigns the union of x and y to x .

Pillar is designed so that a set variable can be treated as a bit vector whose bits correspond to the values in the set type's range; for example, the following code builds a vector whose bits denote divisibility by 4:

```
VARIABLE x : SET [1..n];  
FOR i = 1 TO n LOOP  
  x[i] = (i MOD 4 == 0);  
END LOOP i;
```

There are two ways to assign a proper value to an entire set:

- Assign values to all of the individual elements in the set.
- Assign a value to the entire set.

5.6 Flexible Types

In general, any type-constructor except an enumerated-type-constructor or subrange-type-constructor can contain nonconstant expressions; if one does, it actually defines a family of distinct types, one family member for each possible set of nonconstant values on which the type depends. For example, if variables M and N are declared in an outer block, then consider:

```
VARIABLE x : ARRAY[1..M] OF SET [0..2*N];
```

This declaration can occur only in a local-block, since M and N must be declared in a containing scope. M and N are called *free extents*³

In any particular activation of its block, the variable x technically has one of a family of distinct unnamed array types in which the family members can be indexed by pairs (m,n) , where m ranges over all possible values of M , and n ranges over all possible values of N .

Pillar also provides *flexible types* as the preferred way to describe a family of types. A flexible type is a type that is declared with parameters (called *extent parameters*). Such a type defines a family of types. A flexible-type-declaration declares an identifier as the name of a flexible type.

■ flexible-type-declaration

```
[ TYPE ] identifier ( extent-parameter-declaration-list ) :
```

```
{  named-type  
  set-type-constructor  
  array-type-constructor  
  record-type-constructor } ;
```

The keyword **TYPE** can be omitted only if this type declaration immediately follows another type declaration.

If the flexible-type-declaration contains a named-type, this type must be a bound-flexible-type or a pointer to a bound-flexible-type.

Pillar also provides primitive flexible types, for example **STRING**, which defines this family of types:

```
STRING(0)—strings of length zero  
STRING(1)—strings of length one  
STRING(2)—strings of length two
```

³ As Pillar is defined, free extents do not appear in any language rules. In particular, they do not enter into the question of whether two instances of a type with free extents are equivalent, because the two instances will always have identical values of their free extents.

To illustrate the concept of flexible types, consider rewriting the above declaration of x as a flexible type and a variable of that type:

```
TYPE
  t(j,k: integer): ARRAY [1..j] OF SET [0..2*k];
VARIABLE
  x: t(m,n);
```

Technically, a flexible type is not a type at all, as the term "type" is used in Pillar. Rather, a flexible type is a combination of a set of n extent parameters (each with an ordinal type) and a family of distinct types indexed by n tuples of extent-parameter values. For example:

```
TYPE vector (n : INTEGER[1..] ) : ARRAY[1..n] OF REAL;
```

The line above declares *vector* as the name of a new flexible type with one extent parameter named n , whose values range from one to the maximum INTEGER value. Each instance of this type is a one-dimensional array type with its lower bound equal to one and its element type REAL. The upper bound of the array type is the value of the extent parameter n ; for example:

```
VARIABLE x : vector (100); VARIABLE y : vector (2*50);
```

The variables x and y have the same type because they have the same value (100) of the extent parameter n .

An item whose type is derived from a flexible type is said to "have a flexible type." Thus, in the preceding example, x and y have the flexible type *vector*. New types can be derived from *vector*; for example:

```
TYPE short_vector (m : [0..100]) : vector (m);
TYPE t : short_vector (50);
```

The extent parameters of a flexible type are declared in an extent-parameter-declaration-list.

■ extent-parameter-declaration-list

$$\left(\left\{ \{ \text{identifier} \} , \dots : \left\{ \begin{array}{l} \text{name} \\ \text{subrange-type-constructor} \end{array} \right\} \right\} ; \dots [;] \right)$$

The identifiers are declared as the names of the flexible type's extent parameters.

The type of an extent parameter is given by a subrange-type-constructor, or by a name that denotes an ordinal type. The ordinal values in an extent parameter's range must lie within the range of the type INTEGER (otherwise, a range violation occurs).

As shown in the syntax, a name or constructor can apply to a list of extent parameters whose names are separated by commas.

If t is the name of a type, then the set of extents that can be named as extents of t includes the following, and nothing more:

- If t is declared by a flexible-type-declaration, the set includes all of the extents declared in t 's extent-parameter-declaration.

- If the declaration of t specifies t 's type using a named-type that is a bound-flexible-type (see Section 5.6.1) or a name (of a type) that names the type $t1$, then the set also includes all the names of $t1$'s extents, except names that already appear in the set as a result of the application of the rule above.

5.6.1 Bound Flexible Types

A bound-flexible-type is a binding of a flexible type, and denotes one particular member of a family of types denoted by a flexible type.

■ bound-flexible-type

$$\text{name (} \left\{ \begin{array}{c} \text{expression} \\ * \end{array} \right\} \text{ , ...)}$$

The name must denote a flexible type, f . The number of asterisks and expressions must equal the number of f 's extent parameters. The exceptions to this rule (discussed below) involve captured record extents. Each asterisk or expression is interpreted to yield a value for the corresponding extent parameter. Let t be the instance of f corresponding to the n tuple of extent values. The bound-flexible-type denotes the type t or POINTER t . For example, `s20` below denotes a type that is that particular instance of the type *string* with 20 characters:

```
TYPE
s20: string(20);
```

When an extent value is given as an expression, the expression is interpreted with the corresponding extent parameter's type as the target type. When an extent value is given as an asterisk, it is called a *matching extent*. The value of the matching extent is derived from the type of another item. The use of matching extents is allowed only in the type of a parameter or in a blank_DATA type in type casting. (The matching rules are contained in Section 13.2.5 and Section 9.10.)

In bound-flexible-types that occur in declarations, any expressions must be simple-expressions (these are defined in Section 10.2.1).

A flexible record type can be specified as having its extent parameters captured. This means that the value of the extents for a particular instance of the type are stored in the record. With two exceptions, captured extent values are never specified in a bound-flexible-type, because the extents are already present in the item. The only exceptions to this rule are:

- A bound-flexible-type that is used as the type-specification of a VALUE or VARIABLE declaration
- A bound-flexible-type that immediately follows EXTENDS in a record-type-constructor (see Section 5.11.5)

In the above cases, since the data item does not yet exist but is created by the declaration, values for all extents (captured or not) must be present in the bound-flexible-type.

5.7 Pointer Types

For every distinct named-type t (primitive or created by a type declaration) that is not a bit-class type, there is an associated pointer type, `POINTER t` ; this is also true of flexible types. Thus, with the flexible type `vector` (from the preceding example), there is also the flexible type `POINTER vector`, and instances of it; for example:

```
VARIABLE q : POINTER vector (100);    ! Points to vectors of size 100
```

In the type `POINTER t` , t is called the *associated type* of `POINTER t` .

The value of a pointer item is a storage address or the predeclared primitive value `NIL`.

5.8 String Types

In Pillar, a string value (or variable) is a finite sequence of character values (or variables) of the same type. Pillar has the flexible string types `STRING` and `VARYING_STRING`.⁴

`STRING` has one extent parameter:

```
LENGTH : INTEGER[0..]
```

`STRING(n)` takes on strings of length n as its values.

Pillar has a convenient bracket notation for accessing an individual element of a string or substring; for example, if s is of type `STRING(n)`, then $s[i]$ refers to the i th character of s and has the type `CHARACTER`.

To refer to a substring of s , use a range-specification within the brackets; for example, $s[2..]$ refers to the substring beginning at the second character and continuing through the last character of s . The type of such a substring is the same as the original, except for the extent value; for this example, the data-reference's type is `STRING($n - 1$)`.

An item of type `STRING` is also called a character string. Pillar has a form of literal for character strings (see Section 2.3.3), a set of character string operations (see Section 10.9), and the primitive flexible types `STRING` and `VARYING_STRING`.⁵

`STRING` is the fundamental character string type. Use it in procedure interfaces and in building other types related to character strings.

`VARYING_STRING` is provided as a type for variables that take on values with a variable length but with a reasonable maximum length. `VARYING_STRING` is a primitive flexible record type with special properties that permit it to be treated as taking on character strings as values. As a record type, its declaration is the same as:

```
TYPE
  varying_string (max_length: integer [0..32767]): RECORD
    length: integer [0..32767] size(word);    ! current length
    body: string (max_length);                ! holds current value
    LAYOUT SIZE(byte,*)
      length;
      body;
    END LAYOUT;
  END RECORD;
```

⁴ Possible future extensions of Pillar (by adding new types) include zero-terminated strings (such as are found in C), and strings in which the characters are larger than one byte.

⁵ Should character strings be restricted to 32767 characters? This is consistent with the declaration of `VARYING_STRINGS`, and is necessary if the compiler generates character string instructions for VAX processors.

\\ The type *VARYING_STRING* appears to be obsolete. Better suited to modern architectures would be:

```
TYPE
  varying_string (max_length: integer [0..] size(longword)): RECORD
    CAPTURE EXTENTS;
    length: integer [0..] size(longword);    ! current length
    body: string (max_length);              ! holds current value
    LAYOUT SIZE(longword,*)
      max_length;
      length;
      body;
    END LAYOUT;
  END RECORD;
```

\\

When a data-reference to a *VARYING_STRING* item, *s*, is interpreted as a value, the reference is treated as a reference to the part of *s*'s body determined by the current length:

```
s.body[1 .. s.length]
```

Thus, the value of *s* can be any character string whose length does not exceed *s.MAX_LENGTH*. The same interpretation applies when the bracket notation is used; for example:

```
s[2..]
```

actually refers to:

```
s.BODY[2..s.LENGTH]
```

In an assignment, $s = v$, *v* must be a character string value. This assignment is equivalent to:

```
s.LENGTH = n;
s.BODY[1..n] = v;
```

where *n* is the length of *v*.

Bracket notation can be used to refer to a substring of a *VARYING_STRING*, just as for *STRING*.

For *OUT* and *IN OUT* procedure parameters that have the string types, there are restrictions on the argument in addition to the requirement that it be compatible with the parameter. These restrictions are discussed in Section 13.2.3.1.

5.9 Blank_DATA Types

Blank_DATA types are primitive flexible types provided by Pillar for the purpose of manipulating data without regard to its true type.

Blank_DATA types consist of a sequence of values or variables of the same type. In this respect, they are similar to character strings. Pillar provides the following predeclared primitive flexible *blank_DATA* types:

Blank_DATA Type	Type of Members of the Sequence
BIT_DATA	BIT
BYTE_DATA	BYTE
WORD_DATA	WORD
LONGWORD_DATA	LONGWORD
QUADWORD_DATA	QUADWORD

The members of a sequence defined by a blank_DATA type are also called *elements*.

Each of these types has one extent parameter:

LENGTH : INTEGER[0..]

If t is one of these flexible types, an instance of it, $t(n)$, takes on data of length n as its values.

To facilitate the treatment of ordinary types as blank_DATA, the blank_DATA types are treated specially in type casting and in procedure interfaces using the CONFORM option on parameters.

The same bracket notation as is used for character strings (see Section 5.8) can be used to reference individual elements or subsequences of blank_DATA items. For example, if b is of type BYTE_DATA(n), then $b[i]$ refers to the i th byte of b and has the type BYTE. (If s had type BIT_DATA, the reference's type would be BIT, and so on for the other blank_DATA types.)

An item with the type BIT_DATA(n) can occur at an arbitrary bit position in storage. For this reason, any instance of BIT_DATA is included in the set of bit-class types, and its use is restricted to record fields and array elements.

5.10 Array Types

An array type represents an array, or sequence of data items. This sequence consists of zero or more data items of the same type. Each data item is called an *element* of the array type.

An array type is characterized by one or more index ranges and an element type. A new array type is introduced by an array-type-constructor.

■ array-type-constructor

ARRAY [{ range-specification } , ...] OF type-specification

The range-specifications are interpreted to yield the array type's *index ranges*. The ordinal values in an index range must all lie within the range of the type INTEGER (otherwise, a range violation occurs). The number of ranges is the "number of dimensions" of the array type. The type-specification is interpreted to yield the array type's "element type."

An array item, x , with element type t , contains a sequence of elements all of type t . If the number of dimensions is n , and there are m_k elements in the k th index range, then the array x contains $m_1 \times m_2 \times \dots \times m_n$ distinct elements. The elements are indexed by n tuples (i_1, \dots, i_n) , where i_k lies in the k th index range.

An element of the array is accessed by an array element reference, $x[i_1, \dots, i_n]$, where the subscript expression i_k specifies a value in the k th index range. Partial subscripting is not allowed, that is, values for all subscripts must be given.

The only operation provided for entire arrays is assignment. An assignment simply copies the entire array from the source to the target variable (two base array types are compatible only if they are of equivalent types). If an element in the source array has a well-defined value, the corresponding element in the target array is assigned that value. There is no requirement that all of the source array's elements have well-defined values.

5.11 Record Types

A record represents a collection of *fields*, that is, many data of potentially different types. A record type is introduced by a record-type-constructor.

■ record-type-constructor

```

RECORD [ CAPTURE EXTENTS ; ] [ EXTENSIBLE ; ]
  [ EXTENDS named-type ; ]
  { field-list
    NOTHING; }
  [ record-layout-option ]
END RECORD ;

```

■ field-list

```

[[ field-declaration
  union
  [variant-part ] ]...

```

A field-list must not be empty.

■ field-declaration

```

{ identifier } , ... : type-specification [ = initializer ] ;

```

The identifiers in a field-declaration are declared as the names of distinct fields, all with the type denoted by the type-specification. The scope of a field-declaration is the entire record-type-constructor (including the record-layout-option). A field name within a flexible record type must not be the same as an extent name of the type, since the scope of an extent-declaration includes the record-type-constructor.

If an initializer is present in a field-declaration, it must be a constant-expression, and the type-specification must not denote a type that, by itself, contains any fields specified as requiring initialization. The initializer provides a value to which the declared fields are initialized in any instance of a data item having the declared record type, or in any such data item initialized with the INITIALIZE_FIELDS built-in function (see Section 10.11.6). The initializer is interpreted with the given type-specification as a target type. Neither an initializer nor a type-specification denoting a type containing fields requiring initialization can be present in a field-declaration that is contained in a variant, or be present in more than one alternative of a union.⁷

⁷ Since this disallows some types from being used for fields in variants, Pillar might provide a way to declare an uninitialized type that is otherwise the same as an initialized one.

Records can be declared with a record-layout-option to control their representation in storage (see Section 5.17.3).

5.11.1 Captured Extents

A record-type-constructor that occurs in a flexible-type-declaration can be specified with the keywords `CAPTURE EXTENTS`. This causes all of the type's extents to be fields of the record type. Since the extents themselves determine what instance of the flexible type a particular record represents, they must have values (hence, be *captured*) for the record to represent a valid type. Such a record has its captured extents set to particular values in one of the following ways:

- The compiler allocates a new instance of a record with captured extents. This occurs in the interpretation of a declaration of a data item whose type has captured extents.
- The built-in function `INITIALIZE_FIELDS` (see Section 10.11.6) initializes the captured extents in storage.

Records with captured extents provide a way of declaring self-defining flexible records—records that contain all the information necessary to define their type.

5.11.2 Unions and Variants

A field-list can contain variant-parts and unions, which contain one or more variants (in the case of a variant-part) or alternatives (in the case of a union). If a variant-part or union contains more than one variant or alternative, the same storage can be used for all variants or alternatives. In any record data-item whose type contains a union or variant-part, one alternative or variant is said to be *selected*. The identifier that follows `CASE` is called the *selector*.

Unions and variant-parts are declared using the following syntax:

■ variant-part

```
VARIANTS CASE identifier
{ variant } ...
END VARIANTS;
```

■ variant

```
WHEN { set-of-values } THEN { field-list }
     { OTHERS }           { NOTHING ; }
```

■ union

```
UNION CASE { identifier }
           { * }
{ alternative } ...
END UNION ;
```

■ alternative

```
WHEN { set-of-values } THEN { union-field-list }
     { OTHERS }           { NOTHING ; }
```

■ union-field-list

$$\left\{ \begin{array}{l} \text{field-declaration} \\ \text{union} \end{array} \right\} \dots$$

For both unions and variant-parts, the alternative or variant that is selected is determined by the set-of-values following the WHEN. All range-specifications and expressions in this set-of-values must be constant; they are interpreted with the selector's type as the target type. A field in one of the variants or alternatives is *selected* only if the current value of the selector lies in that variant's or alternative's set-of-values. When a field-reference is interpreted, and it denotes a field in one of the variants or alternatives, that variant or alternative must be selected (otherwise, a range violation occurs).

The specified sets in the set-of-values in the WHEN clauses must not overlap. OTHERS can be used in place of a set-of-values, but for only one variant or alternative. OTHERS denotes the set of all values in the selector's range that are not in any other variant's or alternative's set.

5.11.3 Unions

For a union, Pillar allocates enough storage to contain the largest of the alternatives. A union can occur anywhere in the field list. All fields in a union's alternatives must have constant types (see Section 5.16.5); This ensures that the compiler can treat the entire union as a constant-sized data item.

If an identifier follows the CASE, the union has selected alternatives. The identifier must denote the name of either a field in the record or an extent parameter of the record type that includes the union. (The latter is only allowed if the union is in the record-type-constructor for a flexible record type). If the selector is a field, its declaration must not be in any alternative or variant, and it must precede this union in the record-type-constructor's field-list.

If "CASE *" is used rather than a name following CASE, the alternatives in a union are unselected; there is no checking on correct access to them. When this happens, the values specified following WHEN have no significance in the language, so they are required to be 1 for the first alternative, 2 for the second, and so on. OTHERS is not allowed.

5.11.4 Variants

In contrast to a union, variant-parts are only allocated enough storage to contain one of the variants. Therefore, variants are said to be "selected at allocation." A variant-part can only occur at the end of the field list.

An identifier *must* follow the CASE in a variant-part; an asterisk (*) is not allowed, as it is for unions. The identifier must denote the name of an extent parameter. Therefore, a variant-part can occur only in the record-type-constructor for a flexible record type.

5.11.5 Record Extensions

A record-type-constructor can be declared as being **EXTENSIBLE**, which allows the type to be used as part of a new record type that contains all of the extensible type's fields.

A record-type-constructor must meet the following criteria to be declared as extensible:

- The occurrence of the record-type-constructor must be as the type-specification of a type-declaration.
- The record-type-constructor's field-list must not contain a variant-part.

A record-type-constructor r can be declared using **EXTENDS** named-type. The named-type must denote a record type t that is declared as being **EXTENSIBLE**. The record type defined by r contains, in addition to the fields declared in r , all of t 's fields. Type t 's fields are therefore also in the same scope as r 's fields. The type defined by r is said to *extend* the type t . R begins with t 's fields, and that part of r that is not contained in t is called the *extension*.

The following rules apply to the constructor r :

- The constructor r can contain a record-layout-option only if t is a type with a layout. If r does contain a record-layout-option, it does not affect t 's layout, and does not contain any mention of t 's fields.
- If r is a constructor that defines a flexible record type with captured extents, and the extents of t are also captured, then if er , an extent parameter of r is supplied (in the binding of t following **EXTENDS** in r 's declaration) as the value for et , one of t 's extents, then er does not appear as a field in the extension, since it is already present in t , which is part of r . Therefore, if r contains a record-layout-option, er must not be named therein. The following example illustrates this:

```
TYPE
  rec1(e1: integer): RECORD
    CAPTURE EXTENTS;
    EXTENSIBLE;
    f1a: integer;
    f1b: integer;
    LAYOUT
      e1;
      f1a;
      f1b;
    END LAYOUT;
  END RECORD;

  rec2(e2: integer): RECORD
    CAPTURE EXTENTS;
    EXTENDS rec1(e2);
    f2a: integer;
    f2b: integer;
    LAYOUT
      f2a;
      f2b;
    END LAYOUT;
  END RECORD;
! note that e2 must not appear in rec2's record-layout-option
```

5.12 STATUS

STATUS is a built-in type used in Pillar's condition-handling mechanisms. It is declared in the module \$condition. A value of type STATUS denotes a condition or error that can occur during a program's execution. A STATUS value also has a severity level. Since the type is used in condition handling, it is discussed further in Section 12.2.2.

5.13 ANYTYPE

ANYTYPE is a predeclared primitive type used when dealing with data whose structure cannot be expressed within the Pillar type system. This type defines no valid values, set of operations, or data representation.

ANYTYPE is allowed to occur as a named-type only in contexts for which this is explicitly stated. Unless stated otherwise, a data-reference of type ANYTYPE is not allowed; the data-reference must be type-cast to some other type.

5.14 Procedure Types

Each of the types described heretofore in this chapter is a *data type*, in the sense that a data item of that type can be declared. But in addition to data types, Pillar also provides *procedure types*. Given a procedure type, one can declare a data item of that type, or a procedure of that type.

A procedure type is declared using a procedure-type-declaration:

■ procedure-type-declaration

[TYPE] identifier :

PROCEDURE procedure-type-specification

The keyword TYPE can be omitted only if this type declaration immediately follows another type declaration.

The syntax for a procedure-type-specification is shown in Section 13.1.1.

An example of some declarations of procedure types follows:

```
TYPE
  math_function: PROCEDURE(
                    IN operand: double)
                    RETURNS double;
  sqrt_function: PROCEDURE(
                    operand)
                    OF TYPE math_function;
```

To declare a synonym of a procedure type, the preceding syntax must be used. Note that the following is not allowed:

```
TYPE
  sqrt_function: math_function; ! Not allowed
```


5.15 Opaque Types

An opaque type is a type whose properties (except for its size) are not, in general, made available to users of the type. All of its properties can be made known to a user of the type by *revealing* the type (see Section 4.3).

An opaque type is a way of hiding a type's true nature. The type that is hidden is referred to as the opaque type's *actual type*.

In a scope in which an opaque type *t* is not revealed, only the following uses of the type are permitted:

- Type *t* can be referred to in other declarations (but not in expressions contained in declarations).
- A data item of type *t* can be passed as an argument.
- A data item of type *t* can be used as the target or source of an assignment.
- A data item of type *t* can be typecast to ANYTYPE.

An opaque type is declared either as complete or external, by a complete-opaque-type-declaration or an external-opaque-type-declaration, each of which declares an identifier as the name of an opaque named-type:

■ complete-opaque-type-declaration

[TYPE] identifier OPAQUE [size-option] : type-specification ;

■ external-opaque-type-declaration

[TYPE] identifier OPAQUE [size-option] EXTERNAL ;

Both the complete-opaque-type-declaration and the external-opaque-type-declaration declare the identifier as the name of an opaque type. The complete-opaque-type-declaration also reveals the opaque type in the module in which the declaration occurs.

The use of the size-option is described in Section 5.17.2.

In either the complete or external case, the keyword TYPE can be omitted only if this type declaration immediately follows another type declaration. If the size-option is omitted on an external-opaque-type-declaration, the type's alignment requirement and size is taken to be the same as that for pointer types. An external-opaque-type-declaration must be completed in another module (an implementation module) by an opaque-type-completion:

■ opaque-type-completion

[TYPE] identifier : type-specification ;

The keyword TYPE can be omitted only if this type declaration immediately follows another type declaration.

The identifier must denote an opaque type that was imported in an implement-section of the module containing the opaque-type-completion. An opaque-type-completion also causes the type being completed to be revealed in the containing module.

Opaque types must always have constant-types: the type-specification in a complete-opaque-type-declaration or an opaque-type-completion must denote a constant type (see Section 5.16.5).

5.16 Relationships Among Types

5.16.1 Type Equivalence

Type equivalence is a relationship between two types that means the types have no practical differences (the exact meaning of this is the definition of type equivalence, which follows). Type equivalence is reflexive.

To define type equivalence, the concept of root type is useful. If t is a type, then its root type rt is defined as follows:

- If t is a synonym for $t1$, then rt is the root type of $t1$.
- If t is a binding of the flexible type $t1$, then rt is the root type of the particular instance of $t1$ denoted by the binding. Note that this rule is applied recursively.
- If t is a revealed opaque type (see Sections 4.3 and 5.15) with actual type at , then rt is the root type of at .
- If t is an unrevealed opaque type, then rt is t .
- Otherwise, rt is t .

By these rules, a root type is either a primitive type, a type constructor, or an unrevealed opaque type.

Two types are considered the same if they have identical root types.

Two types are equivalent only if at least one of the following relationships holds between the *root types* of the two types:

- Both are the same nonflexible type.
- Both are the same primitive or record flexible type, and their corresponding extents are equal.
- Both are ordinal types or set types, and both have the same base ordinal type, with their upper and lower bounds equal, their sizes equal, their alignment requirements equal, and both or neither are subrange types.
- Both are array types, and both have the same number of dimensions, the same number of elements in each dimension, and equivalent element types.
- Both are pointer types, and their associated types are equivalent.
- Both are procedure types with all of the following true:
 - Both have the same number of parameters.
 - All their corresponding parameters are equivalent.
 - All their corresponding parameters have the same mode.
 - All their corresponding parameters have identical parameter options and default values.
 - Both or neither must return a value.
 - If both return a value, their return types must be equivalent.
 - Both have the same linkage options.

- Both or neither require an environment. If they require an environment, both must require the same environment.

5.16.2 Type Compatibility

Type compatibility is a relationship between two types meaning that a value of either type can be assigned to an assignable data item of the other type. Type compatibility is reflexive.

Type compatibility is determined by the following rules:

- Two equivalent types are compatible.
- All arithmetic types are compatible.
- Two ordinal types are compatible if they have the same base ordinal type.
- Two set types are compatible if they have the same base ordinal type and the same range.
- All string types are compatible.

Contexts in Pillar that require compatibility provide a *target type* for interpreting an expression or other construction. These contexts are made explicit by such phrases as “The expression is interpreted with the data-reference’s type as its target type.” For example, in $a = b + c$;, the expression $b + c$ is interpreted with the type of a as its target type.

Where compatibility is required and the expression or other construction does not have a type that is equivalent to the target type, the expression’s value is converted to the target type (see Section 5.16.4).

5.16.3 Type Assignment Compatibility

Type assignment compatibility is a *nonreflexive* relationship between two types, meaning that a value of one type can be assigned to an assignable data item of the other type.

The type $t2$ being assignment compatible to the type $t1$ means that a value of type $t2$ can be assigned to a data-reference of type $t1$. This is true if either:

- $t2$ and $t1$ are compatible or
- $t2$ and $t1$ are both pointer types, and the associated type of $t2$ extends the associated type of $t1$

5.16.4 Conversion Between Compatible types

In contexts such as assignment, a source value is implicitly converted to a compatible target type. If the source and target types are equivalent, there is no change in the value or internal representation. If the types are different, there can be a change in value, or the internal representation might change without the value changing.

The following rules cover all cases of implicit conversion between compatible types in which a value can change, or a range violation can occur. (For additional, explicit conversions, see the descriptions of the conversion functions in Section 10.11.5).

Target type: Any integer type
Value's type: Any arithmetic type

If the value is not an integer, it is converted to one by truncating it towards zero. The truncated value must lie in the target type's range (otherwise, a range violation occurs).

Target type: Any floating-point type
Value's type: Any arithmetic type

The value is rounded to the precision of the target type. If the rounded value's magnitude is too large for the target type, an overflow exception occurs. If the rounded value's magnitude is too small for the target type, and underflow is enabled, an underflow exception occurs; if underflow is disabled, the value is converted to zero.

Target type: STRING(*m*)
Value's type: STRING(*n*)

The length *m* must be greater than or equal to the length *n* (otherwise, range violation). If *m* is greater than *n*, the value is padded with enough blanks on the right to make its length equal to *m*. If *m* and *n* are equal, the value does not change. Note that because of the way Pillar is defined, a value used in a conversion never has the type VARYING_STRING.

For STRING OUT and IN OUT parameters, there are restrictions on the argument in addition to the requirement that it be compatible with the parameter (see Section 13.2.3.1).

Target type: VARYING_STRING(*m*)
Value's type: STRING(*n*)

The value's length, *n*, must not exceed the target's maximum length, *m* (otherwise, range violation). The value does not change.

For VARYING_STRING OUT and IN OUT parameters, there are restrictions on the argument in addition to the requirement that it be compatible with the parameter (see Section 13.2.3.1).

5.16.5 Small Types and Constant Types

A type is said to be a *constant type* if its type-specification does not depend on nonconstant expressions.

A type is said to be a *small type* if it is any one of the following:

- Any arithmetic, ordinal, or pointer type
- A constant set type whose range contains at most 64 ordinal values

- A set type with a size-option
- A procedure type
- The type STATUS
- An opaque type declared in an external-opaque-type-declaration either without a size-option, where the size-option does not contain an expression
- An opaque type declared in a complete-opaque-type-declaration with a size-option that does not contain an expression, or whose actual type is small

All small types contain 64 bits or fewer. ⁸

The concept *small type* is used in Pillar language rules because the rules reflect the assumption that small types can reasonably be held in registers and that copying a value of the type is inexpensive.

5.17 Data Representation

Each Pillar data type has an internal representation that specifies how items of the type are normally represented in storage. The compiler is not required to follow these rules in all cases, but it will follow them for data that can be accessed outside of Pillar or through one of Pillar's type escape mechanisms.

The following representational properties of a data type are of special interest. These properties can be explicitly specified for ordinal subrange types, small set types, and record types.

Data Type Property	Description
Size units	Expressed as one of the types BIT, BYTE, WORD, LONGWORD, or QUADWORD.
Size	A nonnegative integer giving the storage size in the specified size units.
Alignment requirement	Expressed as one of the types BIT, BYTE, WORD, LONGWORD, or QUADWORD. Items with this property will be allocated in storage with at least the alignment specified; the lone exception is that fields in a record can be explicitly dealigned (see Section 5.17.3.1).

5.17.1 Standard Data Representation Rules

- The types BIT, BYTE, WORD, LONGWORD, and QUADWORD have the size and alignment requirement implied by their names. The ordinal value is stored as an unsigned integer.
- The type INTEGER has the same size and alignment requirement as LONGWORD. The integer value is stored as a two's complement integer.
- The type LARGE_INTEGER has the same size and alignment requirement as QUADWORD. The integer value is stored as a two's complement integer.
- The type REAL has the same size and alignment requirement as LONGWORD.
- The type DOUBLE has the same size and alignment requirement as QUADWORD.

⁸ Should any other types that contain 64 bits or fewer be considered small types? For example: record types, blank_DATA, strings, and arrays. It is not at all clear that it is appropriate to put these types in registers.

- The type **BOOLEAN** has the same size and alignment requirement as **BYTE**. The values **TRUE** and **FALSE** are represented by their ordinal values: one and zero, respectively.
- The type **CHARACTER** has the same size and alignment requirement as **BYTE**. The ordinal values of the characters listed by Pillar are the values shown in Table 2-1.
- An enumerated type has the same size and alignment requirement as **LONGWORD**.
- Unless a subrange-type-constructor contains a size-option, its internal representation is the same as that of the type from which it is derived.
- The type **blank_DATA(*n*)** has size *n* with size units and alignment requirement as implied by the name.
- If a set-type-constructor does not contain a size-option, then:
 - If the set is constant sized and contains 32 or fewer elements, it has the same size and alignment requirement as **LONGWORD**.
 - Otherwise, it has the same size units and alignment requirement as **QUADWORD_DATA**.

In all sets, the first bit⁹ corresponds to the lowest element in the set's range, and one bit is used to represent each element.

- The type **STRING(*n*)** has the same size and alignment requirement as **BYTE_DATA(*n*)**.
- For the type **VARYING_STRING(*n*)**, the representation is specified by the type's definition as a record type.
- The type **POINTER *t*** has the same size and alignment requirement as **LONGWORD**. Such a pointer item holds a machine address in the form determined by the system. **NIL** is represented by the value zero.
- If, in an array-type-constructor, the element type is a bit-class data type, the array type's size units and alignment requirement are the same as for **INTEGER**; otherwise, they are the same as the element type's. The array's elements are stored in row-major order (which means that the rightmost subscript varies most rapidly, for example, **X[1,1]**, **X[1,2]**,...) with no fill between elements. The only possible fill occurs at the end for a bit-class element type.
- If a record-type-constructor has a record-layout-option, the type's representation is specified by the layout; otherwise, the record's size units and alignment requirement are the same as for **QUADWORD_DATA**.
- In a type-declaration whose type-specification is a named-type (that is, not a constructed type), the representation of the new type is the same as the type from which it is derived.
- The type **STATUS** has the same size and alignment requirement as **QUADWORD**. The quadword contains a system status value.
- All procedure data types have the same size and alignment requirement as pointer types (because they are represented as pointers).

The standard alignment requirement and the size units for a type are the same.

⁹ The meaning of "first bit" is machine dependent. It is the low order bit on VAX machines, but might be different on RISC architectures.

5.17.2 The SIZE Option

A size-option can be given within a subrange-type-constructor that defines a subrange type, within a set-type-constructor that defines a small set type, within the record-layout-option of a record-type-constructor, or in an opaque type declaration. The option controls the size and alignment requirement of the new type.

■ size-option

$$\text{SIZE} \left(\left. \begin{array}{l} \text{BIT} \\ \text{BYTE} \\ \text{WORD} \\ \text{LONGWORD} \\ \text{QUADWORD} \end{array} \right\} \left[\left[\text{, simple-expression} \right] \left[\text{, *} \right] \right] \right)$$

5.17.2.1 Rules for Size-options In Subrange Types

If BIT is specified, the simple-expression (not the asterisk) must occur and must be constant and denote an integer value in the range 1..64. In this case, the subrange type is a bit-class data type.

If BIT is not specified, neither the simple-expression nor asterisk is allowed. The size is a single byte, word, longword or quadword.

If the specified range of ordinal values includes any negative integer values, the ordinal values are represented as two's complement integers; otherwise, they are represented as unsigned integers. The specified size must be large enough to hold all ordinal values in the subrange (otherwise, a range violation occurs).

5.17.2.2 Rules for Size-options In Small Set Types

If BIT is specified, the simple-expression (not the asterisk) must occur and must be constant and denote an integer value in the range 1..64. In this case, the set type is a bit-class data type. If BIT is not specified, neither the simple-expression nor asterisk is allowed. The size is a single byte, word, longword or quadword.

Within the specified size, the first bit corresponds to the lowest element in the set type's range. The specified size must be large enough to hold all elements in the range (otherwise, a range violation occurs).

A set type with a size-option must have constant limits and is limited to 64 elements.

5.17.2.3 Rules for Size-options In Record Types

The size-option is allowed as part of a record layout. BIT cannot be used. The specified size unit (BYTE, WORD, LONGWORD, or QUADWORD) is the alignment requirement for the record type and also the maximum alignment requirement for each field.

Either the asterisk or the simple-expression must occur. The size of the record is first determined by the layout rules. If the asterisk occurs, the size is the minimum number of specified units required to hold the record. If the simple-expression occurs, it must be constant, and must denote an integer value that specifies a size large enough to hold the entire record (otherwise, a range violation occurs).¹⁰

¹⁰ Suggested additional restriction: a constant-expression can only occur if the record type is a constant type.

5.17.2.4 Rules for Size-options in Opaque Types

BIT cannot be specified. The asterisk must not occur. If a simple-expression occurs, it must be constant and denote an integer value. If the expression is omitted, the size is a single byte, word, longword or quadword. In any case, the specified size must be large enough to hold the entire actual type (otherwise, a range violation occurs).

5.17.3 The LAYOUT Option

A record-type-constructor can contain a record-layout-option. A record-layout-option explicitly controls the internal representation of the record by listing all field names in the order in which they are to occur. The exact spacing of fields can be adjusted by filler-components, alignment-options, and position-options. No implicit gaps are allowed; use a filler-component where a field's alignment requirement or position would otherwise cause a gap.

Using a record-layout-option is the only way to get a known record layout. If a record-layout-option is not used in a given record declaration, Pillar is free to choose its own layout, which can include reordering the fields in storage.

■ record-layout-option

```
LAYOUT [ size-option ]  
  {  
    layout-list  
    PACKED IN ORDER  
    ALIGNED IN ORDER  
  }  
END LAYOUT ;
```

If PACKED IN ORDER is specified in a record-layout-option, the compiler supplies a layout which packs each field into a minimal amount of storage. The alignment of each field can be less than its type's standard alignment requirement. The fields are laid out in the same order in which they are specified in the record-type-constructor. ¹¹

If ALIGNED IN ORDER is specified in a record-layout-option, the compiler supplies a layout that uses a minimal amount of storage while still aligning each field in accordance with its type's standard alignment requirement. The fields are laid out in the same order as they are specified in the record-type-constructor.

■ layout-list

```
[ components-layout ] ...  
[ variant-part-layout ]
```

The layout-list must not be empty.

■ components-layout

```
{  
  field-component  
  filler-component  
  union-layout  
}...
```

¹¹ The exact rules for this layout option will be supplied.

■ field-component

identifier ,... [alignment-option] [position-option] ;

All the fields in the record must be named exactly once in field-components within the layout-list. Fields in a variant or alternative must be in the layout-list of a corresponding OVERLAY.

The interpretation of a record layout uses a location counter measuring the current bit offset from the record's origin. The counter's value is increased as the layout-list is processed. At each step, the location counter has a well-defined alignment, which is never greater than the alignment requirement of the entire record.

When a field is processed, it is allocated at the current value of the location counter, whose alignment must equal or exceed the field's alignment requirement (otherwise, a range violation occurs). The location counter is then incremented by the size of the field. If the new location value is constant, the counter's new alignment is the maximum alignment that does not exceed the record type's alignment requirement, and is consistent with the location value (for example, if the location is 16 bits, the maximum consistent alignment is WORD).

If the new location value is not constant, the new alignment is the smaller of the previous alignment and the effective size units of the field. (For a constant-sized field, the effective size units are the maximum units consistent with the actual size.)

The complete interpretation of a record-layout-option is:

- If a size-option is present, it is interpreted. If a size-option is not present, then the record's size units and alignment requirement are the same as for the type QUADWORD, just as they would be if no record-layout-option were present.
- The alignment requirement is determined for the entire record type and for each field.
- The layout-list is processed with an initial location counter value of zero. The initial alignment is equal to the record's alignment requirement.
- To obtain the record size, the final value of the location counter is rounded up to the record's size units. (This might introduce an implicit fill at the end of the record.)

5.17.3.1 Determining Alignment Requirements

If the record-layout-option contains a size-option, the size-option determines the alignment requirement of the record type. This, in turn, determines the maximum alignment possible for each field in the record. Therefore, using a size-option can result in dealigned record fields.

If the record-layout-option does not contain a size-option, the alignment requirement is the same as for the type QUADWORD, just as it would have been if no record-layout-option were present.

If the field-component for a field does not contain an alignment-option, the field's alignment requirement is the minimum of its type's alignment requirement and the entire record type's alignment requirement. If the layout component for a field contains an alignment-option, it specifies the field's alignment requirement; this must not exceed the alignment requirement established by a size-option for the entire record type.

The compiler will generate different code to access an item that is known to be dealigned.

■ alignment-option

$$\text{ALIGNMENT} \left(\left\{ \begin{array}{l} \text{BYTE} \\ \text{WORD} \\ \text{LONGWORD} \\ \text{QUADWORD} \end{array} \right\} \right)$$

Non-bit-class data can never be dealigned to BIT.

5.17.3.2 The Position Option

A position-option is used to specify the exact position of a field from the origin of a record. Only one field can be named in a field-component containing a position-option.¹²

■ position-option

$$\text{POSITION} \left(\left\{ \begin{array}{l} \text{BIT} \\ \text{BYTE} \\ \text{WORD} \\ \text{LONGWORD} \\ \text{QUADWORD} \end{array} \right\}, \text{constant-expression} \right)$$

Whichever units are used in the position-option, it can be considered as specifying the bit offset of the field from the record origin. A position option can only be specified if the current value of the location counter is constant. The constant-expression, if present, must denote an integer value; it supplies the offset value. The offset value must equal the current value of the location counter (otherwise, a range violation occurs); to achieve this, `FILLER(units,*)` can be used. If `FILLER(units,*)` is used, the offset specified in the position must not be less than the filler's location (otherwise, a range violation occurs). The offset value must also be consistent with the alignment requirement of the field (otherwise, range violation); for example, if the alignment requirement is `WORD` then "*offset* MOD 16 = 0" must be true.

5.17.3.3 Filler Components

A filler-component reserves space in the record layout, or indicates the existence of a gap whose size is computed by the compiler from other information. A filler-component has a name (the identifier shown in the syntax below), which is declared with the same scope as a record field. No references to the name are valid.

■ filler-component

$$\text{identifier} : \text{FILLER} \left(\left\{ \begin{array}{l} \text{BIT} \\ \text{BYTE} \\ \text{WORD} \\ \text{LONGWORD} \\ \text{QUADWORD} \end{array} \right\}, \left\{ \begin{array}{l} \text{constant-expression} \\ * \end{array} \right\} \right)$$

¹² A possible extension to the language is a special position-option to allow a field in a record that extends another to occupy the same space as a filler in the extended record type. Records types that can be extended in this manner have *extensible filler*, and their properties are still to be defined. For example, assignments to them are not safe.

The specified units must not exceed the current alignment of the location counter. If the constant-expression occurs, it must denote a positive integer value, and the indicated amount of space is reserved beginning at the current location counter. If an asterisk occurs, the amount of space reserved is the minimum required to be consistent with the alignment requirement and/or explicit position of the next item. All filler-components, including those with an asterisk, must cause some space to be reserved.

5.17.3.4 Variant Part and Union Layouts

Unions and variant-parts are laid out using union-layouts and variant-part-layouts:

■ union-layout

```
UNION [ alignment-option ] [ position-option ]  
{OVERLAY components-layout}...  
END UNION;
```

■ variant-part-layout

```
VARIANTS [ alignment-option ] [ position-option ]  
{OVERLAY layout-list}...  
END VARIANTS;
```

For each variant-part or union in a field-list, the corresponding layout-list must contain a corresponding variant-part-layout or union-layout, and this must contain one "OVERLAY layout-list" or "OVERLAY components-layout" for each nonempty variant or alternative in the variant-part or union. The "OVERLAY layout-list" or "OVERLAY components-layout" must name all the fields in the variant or alternative and no others.

Since in a variant-part, a single variant is selected at record allocation, only that variant is laid out, and only it contributes to the record's size. In a union, however, each alternative is laid out starting with the same initial value of the location counter.

All items in a union's alternative must have constant types (see Section 5.16.5); therefore, at the end of the union-layout, each location counter yields a final location value of the form *initial_value + constant*. The size of the entire union is determined as the maximum of all these constants; the maximum is then treated as a single item in the containing layout-list.

CHAPTER 6

CONSTANTS, LITERALS, AND CONSTRUCTORS

6.1 Constant Declarations

A constant-declaration declares a named compile-time constant. A constant-declaration has no explicit storage, it is *not* addressable. If the constant-declaration is exported, the value of the constant is made available in the compilation of an importing module.

■ constant-declaration

[CONSTANT] identifier **[: type-specification]** = initializer;

A named constant always has a type. If the optional type-specification (which must be constant) is part of the declaration, the type-specification is used as the target type for interpretation of the initializer and is the type of the named constant. Otherwise, the named constant will have the type of the initializer (which must be a constant type) as its type. If the initializer is a set constructor and the type-specification is not present in the constant-declaration, the set constructor must have a named-type. The initializer must *not* be an array- or record-constructor, and must be constant. A named constant must have a type whose type is ordinal, small set, floating, pointer, or STRING. If the named constant's type is string, the string must not have a length longer than 1024. A named-constant *cannot* have a type of VARYING_STRING.

6.2 Literal Constants

As explained in Chapter 2, the natural value of a character-string-literal is a sequence of characters; the natural value of any other form of literal is a number. The rules in this section govern assigning specific Pillar types to occurrences of literals, and interpreting the literal's natural value as required by the target type. Pillar was designed so that a literal will usually get an appropriate type, either by default or because the context provides a target type. In other cases, you can specify a type or use a CONVERT function.

■ literal-constant

$\left. \begin{array}{l} \text{decimal-literal} \\ \text{binary-literal} \\ \text{octal-literal} \\ \text{hexadecimal-literal} \\ \text{floating-point-literal} \\ \text{character-string-literal} \end{array} \right\} \text{ [: named-type]}$

Character-string-literals must have a target type of STRING or VARYING_STRING. All other literals must have a target type that is small (see Section 5.16.5).

6.2.1 Literals with a Named Type

If the named-type is present, the literal is interpreted as a value (see Section 9.2.1) with the named-type as its type. The following are rules for the literals with a named-type:

- The named-type must be constant.
- The named-type must be string, ordinal, integer, floating, small set, or pointer.
- If the literal is a character-string-literal and its length is less than the named-types, the literal will be blank padded.
- With the exception of character-string- and floating-point-literals, if the literal is smaller than the named-type, the literal will be zero extended to the size of the named type.
- If the literal is a decimal-literal, the named-type must be integer or large_integer.
- If the literal is a floating-point-literal, the named type must be REAL or DOUBLE. In addition, the "conversion" of the value must not underflow or overflow.
- If the value of the literal requires a size that is larger than the size of the named-type, it is an error.
- If the named-type is ordinal and the value of the literal is outside the range of the ordinal type, an error occurs.

If a specific type for the literal is not determined by the above rules, its type is determined by default as shown in Table 6-1.

Table 6-1: Default Types for Literal-Constants

Literal-Constant	Default Type
Decimal-literal with value $< 2^{31}$	INTEGER
Decimal-literal with value $\geq 2^{31}$	LARGE_INTEGER
Binary-, octal-, or hexadecimal-literal with value $< 2^{32}$	LONGWORD
Binary-, octal-, or hexadecimal-literal with value $\geq 2^{32}$	QUADWORD
Floating-point-literal	DOUBLE
Character-string-literal with length = 1	CHARACTER
Character-string-literal with length $n > 1$	STRING(n)

A binary-, octal-, or hexadecimal-literal can have a small set type as its target type. In this case, the set value is obtained by considering the binary representation of the literal-constant's natural value; if bit k equals one, the set value contains the k th element in its range. Bits outside the set type's range must be zero; for example:

```
TYPE bitset : SET [0..31];  
VARIABLE x, y : bitset;  
  
x = "3"x;  
y = [0,1];
```

The variables x and y are assigned the same value.¹

¹ The ability to assign a literal without a named-type to a set seems to be of questionable worth. Should this ability be removed?

6.3 Initializers and Constructors

Initializers are used in:

- Value-declarations, to specify the value
- Variable-declarations, to specify an initial value for the variable
- Field-declarations, see Section 5.11
- Parameter-declarations, to specify a default value for an IN parameter
- Array-constructors and record-constructors, to specify the value of an element or field

Array- and record-constructors are always constant, that is, they contain only initializers of NIL, literals, constant-expressions, array-constructors, or record-constructors. Set-constructors used as initializers can have only constant expressions in their set-of-values. However, set-constructors used in expressions in statement-sequences can be dynamic.

6.3.1 Initializers

■ initializer
expression

The type of an initialized item must be constant. The initialized item's type provides the target type for interpreting constructors and literals.

Initialization is sometimes deceptively inefficient; the compiler imposes the following restrictions to eliminate some of the less efficient cases:

- The size of an initialized item must not exceed 65535 bytes.
- If the initialized item is a local variable (that is, not at module level), then its size must not be greater than 1024.

Even in the case of a module level variable, initializing a large variable can involve copying a large amount of data at run time. This data movement can be less efficient than using explicit assignments. If only a few elements of an aggregate are initialized to nonzero values, consider using *OTHERS = "0"x* for array-constructors and *OTHERS = DEFAULT* for record-constructors as the initializer, and then use assignment statements to supply the nondefault values.

6.3.1.1 NIL as an Initializer

When NIL is used as an initializer, the target type must be a pointer type or a procedure type.

6.3.2 Set Constructors

A set-constructor specifies a set value by enumerating the ordinal values contained in the set. The constructor either gives an explicit type for the value, or the context must provide a target type. All values specified in the constructor are required to lie in the type's range.

■ set-constructor

[[set-of-values]] [: named-type]

If the named-type is present, it must denote a set type, *t*; otherwise, the context must provide a target set type, *t*. The range of *t* is taken to specify an ordinal type, and this ordinal type is the target type for interpreting the set-of-values (see Section 6.3.2.1). This set-of-values yields a set value of type *t*. If the set-of-values is omitted, the value is the empty set of type *t*.

The range specified by the set-of-values must be constant if the set-constructor is used as an initializer, but can be variable if the set-constructor occurs as an expression in a local-block's statement sequence.

6.3.2.1 Set of Values

The set-of-values category specifies a set of ordinal values by enumerating one or more ranges or single values that belong to the set. The set-of-values category is used in set-constructors and with the keyword WHEN (the latter in UNION and VARIANTS declarations and CASE statements).

■ set-of-values

{ expression
range-specification }

Each expression or range-specification is interpreted with the ordinal target type provided for the interpretation of the set-of-values (if the context provides such a type). The resulting set contains the ordinal values of the expressions and the ordinal values in the ranges. If the constant does not provide a target type, all the expressions and ranges must have equivalent base ordinal types; this can occur when a set-of-values follows the keyword WHEN in a case-statement or record-type-constructor.

6.3.3 Array Constructors

An array constructor provides a way to specify an array constant.

In an array constructor, the element values can simply be listed in *row-major order*. The same value can be repeated (for example, *10 OF 1* specifies 10 occurrences of the integer 1). The list must provide values for exactly the number of items in the array, unless it uses OTHERS to specify a value for all remaining elements; for example:

{ 1, OTHERS = 0 }

The constructor above is valid for any arithmetic array type that has at least one element. The first element value is one; all others are zero. OTHERS can be used in this case even if there are no remaining elements.

Rather than list all the elements in order, individual elements can be explicitly selected by index values; for example:

```

TYPE
  alkali: (lithium, sodium, potassium, rubidium, cesium, francium);

VALUE
  atomic_number: ARRAY [alkali] OF integer =
    { [lithium] = 3,
      [sodium] = 11,
      [potassium] = 19,
      .
      .
      .
    }

```

There are no rules about the order in which the selected elements occur. All elements must be selected unless OTHERS is used to specify a value for the remaining elements.

The selection notation can be used to select subarrays, which are then given using the positional notation. For example, an identity matrix can be constructed thus:

```

{ [1] = { 1, OTHERS = 0 },           ! first row
  [2] = { 0, 1, OTHERS = 0 },       ! second row
  [3] = { 2 OF 0, 1, OTHERS = 0 }, ! third row
  .
  .
  .
}

```

■ array-constructor

$$\text{"{" } \left\{ \begin{array}{l} \text{element-list} \\ \left\{ \begin{array}{l} \text{selected-element-value} \\ \text{selected-subarray-value} \end{array} \right\} \left[, \text{OTHERS} = \text{initializer} \right] \end{array} \right\} \text{"}$$

If selected-subarray-values are used, they must all have the same dimension. The same element or subarray must not be specified twice by a selected-element-value or selected-subarray-value. If OTHERS occurs, the value of the initializer following OTHERS is used for all other elements of the array; in this case (that is, if selected-subarray-values are used), there must be at least one unselected element or subarray. If OTHERS does not occur, all elements or subarrays must be selected.

■ element-list

$$\left[\left[\left\{ \left[\text{constant-expression OF} \right] \text{initializer} \right\} \dots \left[, \text{OTHERS} = \text{initializer} \right] \right] \right]$$

An element-list specifies the values of an array or subarray in row-major order. Each element's value is specified by an initializer. This initializer's target type is the array's element type.

The constant-expression preceding OF must yield a nonnegative integer; this integer is the number of times the associated initializer's value is repeated in the list.

If OTHERS is used, the number of values must not exceed the number of elements in the target. The value of the initializer following OTHERS is used for any excess target elements. If OTHERS is not used, the number of values in the list must equal the number of elements in the target array type or selected subarray. An element list can be empty if the target array has no elements.

■ selected-element-value

array-selector = initializer

■ selected-subarray-value

array-selector = "{" element-list "}"

■ array-selector

[{ constant-expression } , ...]

An array-selector is used in an array-constructor to select an element or subarray of the target array type. The number, n , of constant-expressions must not exceed the dimension, d , of the array. The k th expression must be compatible with the k th index range of the array.

Let v_k be the value of the k th expression. If n is equal to d , the element with index n tuple $(v_1..v_n)$ is selected. Otherwise, a subarray is selected. The subarray contains all elements with index n tuples $(v_1..v_n, i_{n+1}..i_d)$, where i_k lies in index range k , $k = n+1 ..d$.

6.3.4 Record Constructors

A record constructor provides a way to specify a record constant.

■ record-constructor

"{" [{ identifier = initializer } , ... [, OTHERS = DEFAULT]] "}"

Each identifier must be the name of a field (not a filler) in the record. The corresponding initializer is interpreted with the field's type as its target type.

If OTHERS is used, all fields not named receive default initialization, except for those in excluded variants and alternatives. If OTHERS is not used, all fields in the record type must be named in the constructor, except for those in excluded variants and alternatives. If the record-constructor names a field in a variant, all other variants in the same variant part are excluded; fields in them must not be named in the record-constructor. Similarly, if the record-constructor names a field in an alternative, all other alternatives in the same union are excluded; fields in them must not be named in the record-constructor. A record-constructor can be empty if the record type has no fields. Note that OTHERS can be used even if all fields are named in the initializer; OTHERS has no effect in this case.

6.3.4.1 Using OTHERS In record-constructors

When OTHERS = DEFAULT is used, any target type is allowed. The initial value's internal representation is zero in all bits. Table 6-2 shows the initial value, in Pillar terms, of a field with a given target type when it is initialized with DEFAULT (assuming standard data representation).

Table 6-2: Initial DEFAULT Field Values

Target Type	Initial Value
Arithmetic type	Zero
BOOLEAN	FALSE
CHARACTER	The NUL character
Other ordinal types	Ordinal value zero
Set types	The empty set
STRING	String of NUL characters
VARYING_STRING	Null string
Pointer types	NIL
Procedure types	NIL
STATUS	binary zero
MESSAGE_VECTOR	binary zero

Note that an ordinal item with a subrange type can thus be initialized to a value outside its range.

CHAPTER 7

VALUES AND VARIABLES

7.1 Overview of Values and Variables

This chapter describes declarations of data items that can be declared at block level in Pillar. These declarations are: constant-declarations (see Section 6.1), value-declarations, variable-declarations, bind-declarations, and define-declarations.

A value-declaration causes storage to be allocated for a data item that cannot be modified during its existence. This data item is called a *nonassignable data item*.

A variable-declaration causes storage to be allocated for a data item that can be modified during its existence. This data item is called an *assignable data item*.

A bind-declaration or define-declaration does not cause any storage to be allocated; rather, it renames an already existing data item (or part thereof). A bind-declaration or define-declaration can declare an assignable data item or a nonassignable data item, depending on the properties of the existing data item it renames.

Some of these declarations can or must be declared with an initial value, called an *initializer*. Initializers are described in Section 6.3.1.

7.2 Value Declarations

A value-declaration declares a name for a named value, which is a nonassignable data item with its own location. The value's location is unique across all data items declared by value-declarations and variable-declarations in blocks that are active at the same time.

There are two forms of value-declaration:

■ complete-value-declaration

[VALUE] identifier : type-specification = initializer ;

■ external-value-declaration

[VALUE] identifier : type-specification EXTERNAL ;

The keyword VALUE must be present unless the value-declaration immediately follows another value-declaration or value-completion.

The identifier is declared as the name of a nonassignable data item. The type-specification determines the name's type. Note that a value-declaration does *not* declare a named constant, but it does declare a named value.

In a complete-value-declaration, the initializer is interpreted with the name's declared type (provided by the type-specification) as its target type; the resulting value is the value of the data item.

In an external-value-declaration (which can only occur at module-level), the name's value and storage must be given by a value-completion in the module-level block of an implementation module, or the name's value and storage can be supplied by being implemented in a module written in another language.

The data item defined by a value-declaration exists for as long as its containing block is active. The storage class of a value data item is given in Section 8.1.

7.2.1 Value Completions

A value-completion provides the value and the storage of a named value declared by an external-value-declaration. A value-completion is allowed only at module level; its name must be specified as an implemented name (in one of the module's implement-sections, either explicitly or using wildcard notation). The value-completion is not itself a declaration of the name.

■ value-completion

[VALUE] identifier = initializer ;

The keyword **VALUE** must be present unless the value-declaration immediately follows another value-declaration or value-completion.

The initializer is interpreted with the name's declared type (the name denoted by the identifier) as its target type, and the name denotes the resulting value.

7.3 Variable Declarations

A variable-declaration declares an identifier as the name of a variable, which is an assignable data item with its own location. The variable's location is unique across all data items declared by value-declarations and variable-declarations in blocks that are active at the same time.

There are two forms of variable-declaration:

■ complete-variable-declaration

[VARIABLE] { identifier } , ... : type-specification **[= initializer]** **[[ALIASED] [SHARED]]** ;

■ external-variable-declaration

[VARIABLE] { identifier } , ... : type-specification **EXTERNAL** **[[ALIASED] [SHARED]]** ;

The keyword **VARIABLE** must be present unless the variable-declaration immediately follows another variable-declaration or variable-completion.

If a complete-variable-declaration contains an initializer, the initializer is interpreted with the name's declared type (provided by the type-specification) as its target type; the resulting value is the initial value of the declared variable.

An external-variable-declaration (which can only occur at module level) declares a variable whose storage (and possibly initial value) must be given by a variable-completion in the module-level block of an implementation module, or can be supplied by being implemented in a module written in another language.

The data item defined by a variable-declaration exists for as long as its containing block is active. The storage class of a variable data item is given in Section 8.1.

The keywords SHARED and ALIASED are only allowed on variable-declarations at module level. Their meanings are defined in Section 8.2.

7.3.1 Variable Completions

A variable-completion provides the storage (and possibly initial value) for a name declared by an external-variable-declaration. A variable-completion is allowed only at module level; its name must be specified as an implemented name (in one of the module's implementations, either explicitly or using wildcard notation). The variable-completion itself is not a declaration of the name.

■ variable-completion

[VARIABLE] identifier [= initializer] ;

The keyword VARIABLE must be present unless the variable-declaration immediately follows another variable-declaration or variable-completion.

The initializer, if present, is interpreted with the name's declared type (the name denoted by the identifier) as its target type, and the resulting value is the initial value of the declared variable.

7.4 BIND Declarations

A bind-declaration declares a new data item (called a *BIND item*) that does *not* have a unique location, but has the same location or value as an existing data item.

■ bind-declaration

[BIND] identifier = data-reference ;

The keyword BIND must be present unless the bind-declaration immediately follows another bind-declaration.

The data-reference (see Chapter 9) is interpreted *at the entry to the local-block at whose level it is declared* to yield a location or value. The location or value of the BIND item is the location or value obtained by interpreting the data-reference. The BIND item is assignable only if the data-reference is assignable.

The identifier is declared as the name of a BIND item, which denotes the assignable or nonassignable data item denoted by the data-reference.

Note that the data-reference in a bind-declaration can contain expressions. These expressions need not be simple, unlike expressions in declarations in general.

Bind-declarations are not allowed at module level; they are allowed only at local-block level. The value of a BIND item must not be referred to in another declaration in the same local-block, unless the referring declaration is another bind-declaration or a define-declaration. Therefore, in the following example the declaration of *x* is valid, but that of *y* is *not* valid because it is a non-bind-declaration that refers to the value of *a*:

```
BIND a = p.table;
BIND x = a[f(n)]; ! f(n) is not simple; this is all right.
VALUE y = a[1];
```

One particular fact about bind-declarations should be noted: because of when a bind-declaration is interpreted, a bind-declaration *captures* the location (if its data-reference has a location) or value (if its data-reference has no location) of its data-reference. (A bind-declaration does *not*, however, capture the *value of the location* of its data reference.) That is, the values of all the expressions on which the data-reference depends are captured (remembered) and cannot change while the block (in which the bind-declaration is declared) is active. Therefore, in the following example, if *i* has the value 2 in a given call to *p*, the bind-declaration *x* refers to *a[2]*, even after *i*'s value is modified:

```
PROCEDURE p(IN OUT i: integer);
VARIABLE
  a: ARRAY [1..10] of integer;
BIND
  x = a[i];
BEGIN
  i = i + 1;
  .
  .
  .
END p;
```

7.5 DEFINE Declarations

A define-declaration declares a *DEFINE* item that does *not* have a unique location, but has the same location or value as an existing data item. Each interpretation of a DEFINE item yields a possibly different data item.

■ define-declaration

```
[ DEFINE ] identifier = data-reference ;
```

The keyword DEFINE must be present unless the define-declaration immediately follows another define-declaration.

The data-reference (see Chapter 9) must be a simple reference; it is interpreted *whenever the DEFINE item is encountered in one of the following contexts*:

- A statement
- A non-define-declaration
- The *interpretation* of another DEFINE item

The data-reference is interpreted, in these contexts, to yield a location or value. The location or value of the DEFINE item (which can change with each interpretation of the DEFINE item) is the location or value obtained by interpreting the data-reference. The DEFINE item is assignable if and only if the data-reference is assignable.

The identifier is declared as the name of a DEFINE item, which denotes the assignable or nonassignable data item denoted by the data-reference.

Note that the data-reference in a define-declaration can contain expressions. In contrast to expressions in bind-declarations, the expressions in define-declarations must be simple expressions. In addition, the expressions in module-level define-declarations are *not* restricted to being constant expressions.

The value of a DEFINE item must not be referred to in another declaration in the same local-block, unless the referring declaration is another define-declaration or a bind-declaration. Therefore, in the following example, the declaration of *y* is *not* valid because it is a non-define-declaration that refers to the value of *a*:

```
DEFINE a = p.table;  
VALUE y = a[1];
```

One particular fact about define-declarations should be noted that contrasts them with bind-declarations: because of when a define-declaration is interpreted, a define-declaration does *not* capture the location or value of its data-reference. That is, the values of all the expressions the data-reference depends on are *re-evaluated* every time the define-declaration is used; they can therefore change while the block (in which the define-declaration is declared) is active. Therefore, in the following example, if *i* has the value 2 in a given call to *p*, the define-declaration *x* refers to *a[2]* before *i*'s value is modified, but it refers to *a[3]* after *i*'s value is modified:

```
PROCEDURE p(IN OUT i: integer);  
  VARIABLE  
    a: ARRAY [1..10] of integer;  
  DEFINE  
    x = a[i];  
  BEGIN  
    .  
    .  
    i = i + 1;  
    .  
    .  
  END p;
```

CHAPTER 8

STORAGE ALLOCATION

8.1 Storage Classes

A data item has a *storage class*, which is *automatic*, *static*, *readonly*, or *dynamic*.

Automatic storage is associated with the invocation of a local-block. A data item with automatic storage has a different address in each active invocation of the local-block in which the data item is declared.

A data item with static storage has the same address throughout the execution of a process, and can be modified. Programs that use static storage are not reentrant.

A data item with readonly storage has the same address throughout the execution of a process, but cannot be modified.

Dynamic storage is allocated by the programming environment, not by Pillar, and can only be accessed through pointers.

The storage class of a declared data item is determined by one of the following rules.

- A variable declared in a local-block has automatic storage. A variable declared at module level has static storage.
- A named value declared in a local-block has readonly storage if it is initialized to a constant value, and has automatic storage otherwise. A named value declared at module level has readonly storage.
- The members of an environment have automatic storage associated with local-blocks that enable the environment.
- Named constants do not have a storage class.
- Procedures do not have a storage class.
- An IN, OUT, or IN OUT local parameter or a local result parameter can have automatic storage, or it can have the storage class of the corresponding argument.
- A BIND local parameter has the storage class of the corresponding argument.
- A BIND item has the storage class of the data item to which it is bound.
- A DEFINE item has the storage class of the data item it denotes. \\ \\ *Each use of a DEFINE variable can denote a different data item.* \\ \\

8.2 Data Sharing and Aliasing

Pillar has rules about data sharing that reflect assumptions the compiler can make about data access.

- A data item passed to an IN parameter of a procedure must not be modified during the invocation of the procedure, either through a pointer or through a declaration outside the procedure.
- A data item passed to an OUT or IN OUT parameter of a procedure must not be accessed within the procedure except through the parameter declaration.
- A Pillar variable whose address is made available to Pillar from code written in another language must be declared at module level with the ALIASED option.
- A variable declared at module level that can be accessed in more than one thread must be declared with the SHARED option.
- If a data item that can be modified can be accessed in more than one thread, access to it must be through an atomic operation or must be within a guarded critical region.

8.3 Environments

Environments provide a mechanism for global variables without static storage. Proper use is to declare an environment whose members are a program's global variables, enable the environment in the main procedure of the program, and supply the environment to all procedures that reference global variables. If a program is organized into subprograms, each subprogram can organize its global variables into an environment that extends the environment of the main program, and enable its environment in its main procedure.

8.3.1 Environment Declarations

■ environment-declaration

```
ENVIRONMENT identifier [ EXTENDS (name) ] ;  
    [member-variable-declaration]...  
END ENVIRONMENT ;
```

An environment-declaration can only occur at module level. The identifier is declared as an *environment*.

If EXTENDS occurs in the declaration of an environment *env*, the name must denote an environment *env1*, and all of the members of *env1* become members of *env* (while remaining members of *env1*). *Env* is said to *extend env1*, and also to extend any environments that *env1* extends.

■ member-variable-declaration

```
{ identifier } ,... : type-specification ;
```

Each identifier is declared as a member of the enclosing environment, and has the type obtained by interpreting the type-specification.

8.3.2 Properties of Environments

Environments are not data items, but their members are. The declaration of an environment does not allocate storage for its members. Storage for the members of an environment is allocated by beginning the execution of a block that enables the environment.

An environment is *accessible* within the main statement-sequence and the handler sections of a block that enables the environment, and within the body of a procedure that requires the environment. Only if an environment is accessible is it possible to:

- Refer to a member of the environment using a dot-qualified-reference
- Refer by name to a procedure that requires the environment
- Enable an environment that extends the environment

8.3.3 Enabling an Environment

The enable-section of a local-block *b* can enable an environment *env*.

- *Env* must not be accessible at entry to *b*.
- If *env* extends environment *env1*, *env1* must be accessible at entry to *b*.

The effect is to allocate automatic storage for the members of *env*, and to make *env* accessible within the main statement-sequence and handler sections of *b*.

8.3.4 Procedures and Environments

A procedure type *t* can specify that a procedure of type *t* requires an environment. If procedure *p* requires environment *env*, then:

- *Env* is accessible within the body of *p*, as are any environments that *env* extends.
- *Env* must be accessible at each reference to *p*.



CHAPTER 9

DATA REFERENCES

This chapter describes Pillar data references.

9.1 Syntax of Data References

A data reference refers to data.

■ data-reference

{	name
	procedure-function-reference
	built-in-function-reference
	indirect-reference
	dot-qualified-reference
	indexed-element-reference
	substring-reference
	type-cast-reference

9.2 Interpretation of References

The interpretation of a data-reference (also called simply a *reference*) yields a typed *location* or *value*; that is, a reference *ref* denotes location *loc* or value *val* of type *t*.

9.2.1 Locations and Values

A location *loc* is an address with a type *t*.

- If *t* is a record type, and *f* is a field of *t* with offset *o* and type *ft*, field *f* of *loc* is the location *loc* + *o*, with type *ft*.
- If *t* is an array, string, blank_DATA, or set type with element type¹ *et*, element *n* of *loc* is the location *loc* + the offset of element *n* in *t*, with type *et*.

A location is *assignable* or *nonassignable*. If a location is assignable, its elements or fields are also assignable. If a location is nonassignable, its elements or fields are also nonassignable.

A value *val* is data with a type *t*.

- If *t* is a record type, and *f* is a field of *t* with offset *o* and type *ft*, field *f* of *val* is the data in *val* at offset *o*, with type *ft*.

¹ The element type of a string type is CHARACTER; the element type of a set type is BIT.

- If t is an array, string, blank_DATA, or set type with element type et , element n of loc is the data in val at the offset of element n in t , with type et .

9.2.2 The Value of a Reference Rule

If a reference ref that denotes a location loc of type t occurs in a context requiring a value, loc is converted to a value val of type $t1$ by the following rules:

- If t is *ANYTYPE*, ref is in error.
- If t is *VARYING_STRING*, val is the data at the *body* field of loc , and $t1$ is *STRING*(n), where n is the value of the *length* field of loc .
- Otherwise, val is the data at loc , and $t1$ is t .

If a reference ref that denotes a value val with type t occurs in a context requiring a value, val is converted to a value $val1$ of type $t1$ by the following rules:

- If t is *VARYING_STRING*, $val1$ is the *body* field of val , and $t1$ is *STRING*(n), where n is the *length* field of val .
- Otherwise, $val1$ is val , and $t1$ is t .

9.3 Reference to a Named Data Item

A name can occur as a data reference if it names a data item. Any use of a name as a data reference not assigned a meaning by one of the following sections is in error.

9.3.1 Reference to a Named Constant

Interpretation of a reference to a named constant c yields the value of c , with the type of c .

9.3.2 Reference to an Element of an Enumerated Type

Interpretation of a reference to element e of enumerated type t yields the value of e , of type t .

9.3.3 Reference to a Named Value

Interpretation of a reference to a named value v yields the location of v , which is nonassignable, with the type of v .

9.3.4 Reference to a Variable

Interpretation of a reference to a variable v yields the location of v , which is assignable, with the type of v .

9.3.5 Reference to a Bind Item

Interpretation of a reference to a BIND item v yields the location or value to which v is bound, with the type of the location or value to which v is bound.

9.3.6 Reference to a Define Item

Interpretation of a reference to a DEFINE item v yields the result of interpreting the reference that is the definition of v .

9.3.7 Reference to a Loop Control Variable

Interpretation of a reference to a loop control variable i yields the value of i , with the type of i . \ \ \ Note that the name of the control variable is not interpreted as a reference during the initialization of a loop. \ \ \

9.3.8 Reference to an IN Parameter

Interpretation of a reference to an IN parameter p of a procedure type t within the parameter list of t , which can only occur during the interpretation of an invocation of a procedure of type t , yields the value of the argument corresponding to p , with the type of p . Parameter p must not be LIST or OPTIONAL, and must have an ordinal type.

9.3.9 Reference to an IN Local Parameter

Interpretation of a reference to an IN local parameter arg of procedure p within the body of p , yields the value of arg , with the type of arg . Parameter arg must not be LIST. A range violation occurs if arg is OPTIONAL and not present.

9.3.10 Reference to an OUT, IN OUT, BIND, or Result Local Parameter

Interpretation of a reference to an OUT, IN OUT, or BIND local parameter, or a local result parameter arg of procedure p , within the body of p yields the location of arg , which is assignable, with the type of arg . Parameter arg must not be LIST. A range violation occurs if arg is OPTIONAL and not present.

9.3.11 Reference to a Procedure

Interpretation of a reference to a procedure p yields the value of p with the type of p . The value of p is closed over the current activation of p 's parent local-block if p is a subprocedure, and over p 's environment if p requires an environment.

9.3.12 Reference to a Condition

Interpretation of a reference to a condition c yields a value s , of type *STATUS*. The severity of s is the same as the severity of c , and the condition of s is c .

9.4 Reference to the Value of a Procedure Invocation

- procedure-function-reference
 - procedure-invocation

Interpretation of a procedure-function-reference yields the result of interpreting the procedure-invocation, which must produce a value.

9.5 Reference to the Value of a Built-in Function Invocation

■ built-in-function-reference

built-in-function-invocation

■ built-in-function-invocation

name ([argument-list])

The name denotes a built-in function. Interpretation of a built-in-function-reference yields the result of interpreting the built-in-function-reference as an expression.

9.6 Indirect Reference

■ indirect-reference

data-reference[^]

The interpretation of an indirect-reference ptr_ref^{\wedge} proceeds as follows. Ptr_ref is interpreted to obtain a value ptr_val of type pt . Pt must be a pointer type, with associated type t . Interpretation of the indirect-reference yields the location loc of type loc_type obtained by dereferencing ptr_val .

9.6.1 Dereferencing a Pointer Value

Dereferencing a value ptr_val of a pointer type with associated type t yields a location loc of type loc_type by the following steps:

- Loc is the location given by ptr_val . A range violation occurs if ptr_val is NIL. An error with unpredictable consequences occurs if ptr_val is an invalid address.
- Loc_type is obtained from t and loc , as follows:
 - If t has captured extents, values for the extents are obtained by referring the data at loc . A range violation occurs if an obtained value is inconsistent with the type of the corresponding extent. Loc_type is t with the obtained extent values.
 - Otherwise, loc_type is t .

9.6.2 Implicit Dereferencing

A location or value $base_ref$ of type t can be implicitly dereferenced to yield $base_ref1$ (which can be a location or value) of type $t1$ by the following rule:

- If t is a pointer type, ptr is the value of type pt obtained by applying the value of a reference rule to $base_ref$, and $base_ref1$ is the location of type $t1$ obtained by dereferencing ptr .
- Otherwise, $base_ref1$ is $base_ref$, and $t1$ is t .

9.7 Dot-qualified Reference

There are two forms of dot-qualified-references. The first is the general form; the second is the special form:

■ dot-qualified-reference

$$\left\{ \begin{array}{l} \text{data-reference.identifier} \\ \text{name.identifier} \end{array} \right\}$$

A dot-qualified-reference *decl_name.i* is of the special form if *decl_name* denotes a type, a parameter, a LIST parameter, a local parameter, or an environment; otherwise, it is of the general form.

The interpretation of a dot-qualified-reference *ref.i* of the general form proceeds as follows:

1. *Ref* is interpreted to obtain *base_ref* (which can be a location or a value), of type *t*.
2. If *t* is a bound-flexible-type, and *i* is the name of an extent of *t*, the dot-qualified-reference is interpreted as a reference to extent *i* of *t*.
3. *Base_ref* is implicitly dereferenced to obtain *base_ref1* of type *t1*.
4. If *t1* is a bound-flexible-type, and *i* is the name of an extent of *t1*, the dot-qualified-reference is interpreted as a reference to extent *i* of *t1*.
5. If *t1* is a record type, and *i* is the name of a field in *t1*, interpretation of the dot-qualified-reference yields the field *i* of *base_ref1*, with the type of *i* in *t1*.
6. If none of the previous steps yields a result, the dot-qualified-reference is in error.

The interpretation of a dot-qualified-reference *decl_name.i* of the special form depends on what *decl_name* denotes. Any dot-qualified-reference of the special form not assigned a meaning by one of the following sections is in error.

9.7.1 Reference to an Element of an Enumerated Type

If, in *enum.i*, *enum* denotes a root enumerated type *t*, and *i* is a name of an element of *t*, interpretation of the dot-qualified-reference yields the value of *i* in *t*, with type *t*.

9.7.2 Reference to an Extent of a Named Type

If, in *type_name.i*, *type_name* denotes a bound flexible type *t*, and *i* is the name of an extent of *t*, interpretation of the dot-qualified-reference yields the value of *i* in *t*, with the type of *i* in *t*.

9.7.3 Reference to an Extent of a Parameter

If, in *param.i*, *param* denotes a parameter with bound flexible type *t* of a procedure type *pt*, and *i* is the name of an extent of *t*, the dot-qualified-reference is interpreted as a reference to extent *i* of *t*. *Param* cannot be LIST or OPTIONAL. Interpretation of a reference to an extent of a parameter of a procedure type *pt* can occur only during the interpretation of an invocation of a procedure of type *pt*.

9.7.4 Reference to the Length of a LIST Parameter or Local Parameter

If, in *list_param.i*, *list_param* denotes a LIST parameter or local parameter and *i* is LENGTH, interpretation of the dot-qualified-reference yields a value equal to the number of elements in *list_param*, with the type INTEGER [0 ..].

9.7.5 Reference to an Element of an Environment

If, in *env.i*, *env* denotes an accessible environment and *i* is the name of an element of *env*, interpretation of the dot-qualified-reference yields the location of *i* in *env*, with the type of *i* in *env*.

9.8 Indexed Reference

There are two forms of indexed-element-references. The first is the general form, the second, the special form.

■ indexed-element-reference

$$\left\{ \begin{array}{l} \text{data-reference [\{ expression \} , ...] } \\ \text{name [expression]} \end{array} \right\}$$

An indexed-element-reference *decl_name[expression]* is of the special form if *decl_name* denotes a LIST local parameter; otherwise, it is of the general form.

The interpretation of an indexed-element-reference *refl [expression] , ...]* of the general form proceeds as follows:

1. *Ref* is interpreted to obtain *base_ref* (which can be a location or a value), of type *t*.
2. *Base_ref* is implicitly dereferenced to obtain *base_ref1* of type *t1*.
3. One of the following cases applies:
 - If *t1* is an array type with *n* dimensions and element type *et*, there must be exactly *n* expressions. For each dimension *k*, the array type contains a range *r_k*, and the *k*th expression is interpreted with *r_k* as a target range, resulting in an *n*-tuple of values that selects an element *e* of *baseref1*. Interpretation of the indexed-element-reference yields *e*, of type *et*.
 - If *t1* is a set type with range *r*, there must be exactly one expression, which is interpreted with *r* as a target range to obtain the value *v*. Interpretation of the indexed-element-reference yields the element of *base_ref1* that represents containment of *v*, of type *BIT*.
 - If *t1* is *STRING(n)* or *blank_DATA(n)*, there must be exactly one expression, which is interpreted with *INTEGER[1 .. n]* as a target type to obtain a value *v*. Interpretation of the indexed-element-reference yields the *v*th element of *base_ref1*, of the element type of *t1*.
 - If *t1* is *VARYING_STRING(n)*, there must be exactly one expression, which is interpreted with *INTEGER[1 .. n]* as a target type to obtain a value *v*. Interpretation of the indexed-element-reference yields the *v*th element of the *body* field of *base_ref1*, of type *CHARACTER*.
 - Otherwise, the indexed-element-reference is in error.

The interpretation of an indexed-element-reference *decl_name[expression]* of the special form depends on the what *decl_name* denotes. Any indexed-element-reference of the special form not assigned a meaning by one of the following sections is in error.

9.8.1 Reference to an Element of a LIST Local Parameter

If, in *list_arg[expression]*, *list_arg* denotes a LIST local parameter of length *n*, the *expression* is interpreted with INTEGER [1 .. *n*] as a target type to obtain a value *v*. Interpretation of the indexed-element-reference yields the *v*th element of *list_arg* (which is a value if *list_arg* is IN and a location otherwise), with the type of *list_arg*.

9.9 Substring Reference

■ substring-reference

data-reference [range-specification]

The interpretation of a substring-reference *ref[range_spec]* proceeds as follows:

1. *Ref* is interpreted to obtain *base_ref* (which can be a location or a value), of type *t*.
2. *Base_ref* is implicitly dereferenced to obtain *base_ref1* of type *t1*.
3. Type *t1* must be STRING n , blank_DATA n , or VARYING_STRING n . *Range_spec* is interpreted with INTEGER [1 .. *n*] as a target type to obtain the range INTEGER [*first* .. *last*].
 - If *t1* is STRING n , interpretation of the substring-reference yields a substring of *base_ref1* beginning with element *first*, of type STRING(*last* - *first* + 1).
 - If *t1* is blank_DATA n , interpretation of the substring-reference yields a substring of *base_ref1* beginning with element *first*, of type blank_DATA(*last* - *first* + 1).
 - If *t1* is VARYING_STRING n , interpretation of the substring-reference yields a substring of the *body* field of *base_ref1* beginning with element *first*, of type STRING(*last* - *first* + 1).
 - Otherwise, the substring-reference is in error.

9.10 Type Cast Reference

■ type-cast-reference

data-reference :: { named-type [TRUNCATE] }

The interpretation of a type-cast-reference *ref::[typename [TRUNCATE]]* proceeds as follows:

1. *Ref* is interpreted to obtain *base_ref* (which can be a location or a value), of type *t*. *Base_ref* has alignment *al*, which is derived during the interpretation of *ref* if *ref* is a type-cast-reference, and is otherwise the alignment requirement of *t*.
2. *Typename* is interpreted to obtain the type *t1*, with alignment requirement *al1*.
3. If *t* is a bit-class type, *t1* must be a bit-class type.
4. If *t1* is ANYTYPE, the TRUNCATE option is not allowed. Interpretation of the type-cast-reference yields *base_ref*, with type ANYTYPE and alignment *al*.
5. If *t* is ANYTYPE, *t1* cannot have matching extents, and the TRUNCATE option is not allowed. Interpretation of the type-cast-reference yields *base_ref*, with type *t1* and an alignment of the maximum of *al* and *al1*.
6. Neither *t* nor *t1* may be an unrevealed opaque type.

7. If either t or $t1$ is a procedure type, both must be procedure types, and the *TRUNCATE* option is not allowed. Interpretation of the type-cast-reference yields *base_ref*, with type $t1$ and alignment al .
8. If either t or $t1$ is a pointer type, both must be pointer types, and the *TRUNCATE* option is not allowed. Let at be the associated type of t , and $at1$ be the associated type of $t1$. Then:
 1. If $at1$ is *blank_DATA(*)*, at must have a size (at cannot be *ANYTYPE*). The size of at in the size units of $at1$ replaces the matching extent. A range violation occurs if the size of at is not integral in the size units of $at1$.
 2. If either at or $at1$ is *ANYTYPE*, interpretation of the type-cast-reference yields *base_ref*, with type $t1$ and alignment al .
 3. Neither at nor $at1$ may be an unrevealed opaque type.
 4. Neither at nor $at1$ may be a pointer type.
 5. If either at or $at1$ is a procedure type, both must be procedure types.
 6. If both at and $at1$ are record types, and one extends the other, interpretation of the type-cast-reference yields *base_ref*, with type $t1$ and alignment al .
 7. at and $at1$ must have the same size and alignment requirement.
 8. Interpretation of the type-cast-reference yields *base_ref*, with type $t1$ and alignment al .
9. al must not be less than all .
10. If $t1$ is *blank_DATA(*)*, t must have a size (t may not be *ANYTYPE*), and the *TRUNCATE* option is not allowed. The size of t in the size units of $t1$ replaces the matching extent. A range violation occurs if the size of t is not integral in the size units of $t1$. Interpretation of the type-cast-reference yields *base_ref*, with type $t1$ and alignment al .
11. If both t and $t1$ are record types, and one extends the other, interpretation of the type-cast-reference yields *base_ref*, with type $t1$ and alignment al .
12. If either t or $t1$ is a record type, it must have an explicit layout.
13. If $t1$ has captured extents, its extents are replaced by values obtained from *base_ref*, treating *base_ref* as if it were of type $t1$.
14. A range violation occurs if the *TRUNCATE* option is specified and the size of $t1$ is greater than the size of t , or if the *TRUNCATE* option is not specified and the size of t is not equal to the size of $t1$.
15. Interpretation of the type-cast-reference yields *base_ref*, with type $t1$ and alignment al .

9.11 Simple References

A data-reference sr is a *simple reference* if it has the following properties:

- Sr is not a built-in-function-reference or a procedure-function-reference.
- Any data-reference contained within sr is a simple reference.
- Any expression contained within sr is a simple expression.

CHAPTER 10

EXPRESSIONS

This chapter describes Pillar's expressions and arithmetic, boolean, ordinal, set, character string, pointer, and type operators.

10.1 Syntax of Expressions

■ expression

{
literal-constant
set-constructor
data-reference
(expression)
relational-expression
infix-operator-expression
prefix-operator-expression
array-constructor
record-constructor
NIL
}

The expression's interpretation is the value and type of the contained construction (literal-constant, set-constructor, and so forth). If the expression is interpreted with a target type, so is the contained construction. If the contained construction is a data-reference, the reference is interpreted as a value-producing reference.

■ relational-expression

subexpression {
<
<=
==
<>
>=
>
} subexpression

■ subexpression

expression

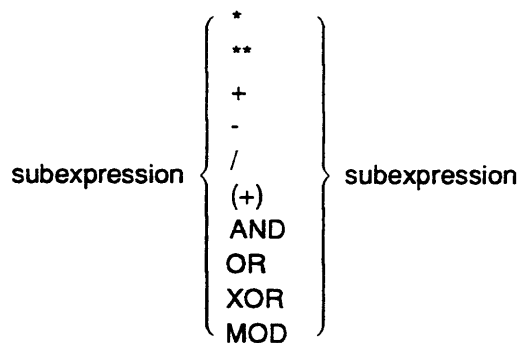
Neither of the subexpressions can be a relational-expression; for example:

`x < y < z ! This is invalid.`

Note: There is no target type for the interpretation of the subexpressions; they can be any kind of expression except a relational-expression.

The rules for relational operations, given later in this chapter, are classified by data type; for example, the character string operator is discussed in Section 10.9.

■ infix-operator-expression



The subexpressions can be any kind of expression.

Pillar's associativity for infix operators is left to right. If the first subexpression is an infix-operator-expression, its operator's precedence must equal or exceed that of the surrounding infix-operator-expression. If the second subexpression is an infix-operator-expression (regardless of whether the first subexpression is), its operator's precedence must *exceed* that of the surrounding infix-operator-expression.

Infix Operator Precedence

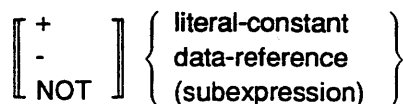
- highest **
- * , / , MOD
- + , - , (+)
- < , <= , = , <> , >= , >
- AND
- lowest OR , XOR

Examples:

Expression	Equivalent Parenthesized Expression
$x + 2 * y$	$x + (2 * y)$
$x + y + z$	$(x + y) + z$
$(-)x / 3 * k$	$(-x / 3) * k$

The rules for infix operations, given later in this chapter, are classified by data type. Note: There is no target type for the interpretation of the subexpressions.

■ prefix-operator-expression



The rules for prefix operations, given later in this chapter, are classified by data type. Prefix operators have higher precedence than infix operators. Note: There is no target type for the interpretation of the subexpression.

10.2 Simple and Constant Expressions

An expression occurring in a Pillar declaration (that is, not as part of an executable statement) must be a simple expression, except when stated otherwise in this manual. Furthermore, if the declaration is at module level, the expression must generally be a constant expression.

The rules for what can occur in simple expressions are summarized in Section 10.2.1. The rules for what can occur in constant expressions are summarized in Section 10.2.3.

There are some places in the syntax where the requirement for a constant or simple expression can be shown explicitly using the constant-expression or simple-expression category.

■ constant-expression

expression

■ simple-expression

expression

10.2.1 Summary of Simple Expression Rules

The restrictions on simple expressions have been chosen so that the expression can easily be computed at compile time when the operands are constants, and so that the class of expressions is sufficient for effective use of Pillar's flexible type facility.

A simple expression can be:

- A simple reference that can be interpreted to produce a value with an ordinal type (see Section 10.2.2 for the rules governing nonconstant values.)
- One of the arithmetic, Boolean, or ordinal operations defined in Sections 10.5 to 10.7, if the operands to the operation are simple expressions
- MAX (*t*) or MIN (*t*), where *t* is a name denoting an ordinal type
- DATA_TYPE_SIZE (*units*, *item*) or FIELD_OFFSET (*units*, *item*, *field-name*), where *item* is a name denoting a type (not a data-item)

10.2.2 Dynamic Values In Simple Expressions

A name in a simple expression can denote a nonconstant value only in the following cases:

- Within the declaration of a flexible type, the name can denote the value of one of the type's extent parameters.
- Within a declaration that is local to a local-block, the name can denote a value known at entry to the local-block. (The value is captured at entry to the local-block.) However, this value cannot be a value depending on an OPTIONAL or LIST parameter of a procedure. *\\\ This does not disallow referring to the length of a local LIST parameter x using the simple reference "x.length" in a simple expression, because there actually is no reference to x in this case. *
- Within a procedure-type-constructor, the name can denote:
 - The value of a non-OPTIONAL, non-LIST IN parameter of the procedure

— The value of an extent (of the flexible type) of a required parameter of any mode

10.2.3 Summary of Constant Expression Rules

If the values referred to in a simple expression are named constants or literals, the simple expression is constant. When an operation applies to a type, the type must be constant.

Expressions that are not simple can be constant if all operands in the expression are constant. \\ \\ *Exact rules TBS. It might be necessary to rule out some operators.* \\ \\

10.3 Principles of Expression Evaluation

In general, the compiler treats an expression as a formula specifying a value, rather than as a sequential computation rule. The compiler's goal is to generate code that will efficiently compute the specified value; Pillar programmers should understand the implications of this regarding expressions that cause exceptions or produce side effects through function calls.

10.3.1 Order of Evaluation

For a typical Pillar construction, the description of the construction's interpretation (as presented in this manual) implies an order for performing operations and interpreting subexpressions. These descriptions might seem to define a strict order for all the operations in a program, but such is not the case. The actual order of evaluation can differ from the implied order. Within a simple statement (one not containing other statements nested within it), the order in which expressions and subexpressions are evaluated is arbitrary. (Of course, the operands of an operator or function call must be evaluated before the operator or function can be evaluated.)

Within a statement-sequence (and any nested statement-sequence), an operation can be performed out of order if the only possible side-effects are changing the particular exception that occurs, or changing the point at which it occurs. This reordering will not cause an exception in a statement-sequence that would otherwise complete without an exception.

Here is an example of order-sensitive code:

```
w = 1/z;  
y[i MOD k] = f(k);
```

The assignment-statement's rules imply that the reference $y[i \text{ MOD } k]$ will be interpreted before the function $f(k)$ is invoked; *this implication is false*. If the function f increments the argument k as a side effect, k can be incremented before or after it is used in the subscript calculation.

If z or k has the value zero, a range violation—most likely a zero-divide exception—occurs. The division $1/z$ is not necessarily performed before $i \text{ MOD } k$ is evaluated. Hence, if both z and k are zero, either a floating- or integer-zero-divide exception can occur. Also, note that if the function f sets k to zero, this can cause an exception, depending upon whether k is set before or after it is used in the subscript calculation.

10.3.2 Incomplete Evaluation

The Pillar compiler might not evaluate a subexpression if its value is not needed to determine the value of a containing expression. This is true even if a function can have a side effect. For example:

```
y = 0;
z = 0;
x = y * (random ( ) + 1/z);
```

The function *random* might not be called (this is true regardless of whether *random* is an inline procedure), and the zero-divide exception resulting from the evaluation of $1/z$ might not occur.

10.3.3 Evaluation of Integer Operations

Integer operations are exact, but overflow can occur; that is, the true result of an expression can be outside the range of the result's type. Pillar does not directly define the cases in which integer overflow causes an exception; instead, integer overflow is treated as an implementation-dependent problem that is to be avoided (by the compiler) if possible. The actual behavior depends on whether range checking is enabled.

If range checking is enabled, the compiler ensures that a range violation (by the true value) is detected explicitly unless an exception, such as overflow, occurs in evaluating the expression. When range checking is disabled, the compiler can transform integer expressions under algebraic laws, and it can use instructions that do not cause integer overflow. If an integer expression's true value is within the range required by its target context, the compiled code will yield the correct value.

The compiler will not transform an expression that does not cause overflow into one that does. Compile-time evaluation of expressions always uses the expression in original form, and overflows are always detected.

10.3.4 Evaluation of Floating-Point Operations

Floating-point operations are performed with overflow enabled. Underflow detection is disabled by default, with underflow producing a result of zero. Underflow detection can be enabled for all expressions in a local-block (see Chapter 12).

As with all exceptions, Pillar treats an arithmetic exception as a serious error. It is impossible to continue execution from the point of exception; exception handling must be performed.

10.4 Interpretation with a Target Type

Interpretation of an expression e with a target type t has the following effects:

- Should the interpretation of e require a target type, t is available.
- The type of e must be assignment compatible to t .
- The value of e is converted to t according to the rules in Section 5.16.4.

Interpretation of an expression e with a target range r has the following effects:

- E is interpreted with the base ordinal type of r as a target type.

- A range violation occurs if the value of e is outside of r .

10.5 Arithmetic Operations

This section describes Pillar's arithmetic operators and built-in functions.

Unless specified otherwise, an operand can have any arithmetic type. The result's type is the common arithmetic type of the operands, and operands are converted to the common type as required. The common type of a set of arithmetic types is the highest of the types of the operands according to the following order:

highest	DOUBLE
	REAL
	LARGE_INTEGER
lowest	INTEGER

For the purpose of determining the common arithmetic type, the base ordinal type of integer types is used.

An arithmetic operation can result in a range violation if the result is out of the target's range. As with any range violation, these range violations will not be detected at run time unless the module is compiled with range checking enabled. Arithmetic instructions that do not detect overflow will be generated by the compiler unless range checking is enabled at compile time.

10.5.1 Negation Operator

$result = -x ;$

10.5.2 Addition Operator

$result = x + y ;$

10.5.3 Subtraction Operator

$result = x - y ;$

10.5.4 Multiplication Operator

$result = x * y ;$

10.5.5 Division Operator

$result = x / y ;$

A range violation (typically a hardware division-by-zero exception) occurs if the value of y is zero. For an integer division, the result satisfies this mathematical relation:

$$x = y * result + r$$

where r is the remainder and satisfies:

$$ABS(r) < ABS(y)$$

$$SIGN(r) = SIGN(x)$$

10.5.6 Integer MOD Operator

$$result = x \text{ MOD } y ;$$

The types of x and y must be integer types. The result has the common type of x and y , and is the integer satisfying:

$$x = m * y + result$$

$$0 \leq result < y$$

where m is an INTEGER or a LARGE_INTEGER.

A range violation occurs if $y \leq 0$.

10.5.7 Arithmetic Comparison Operators

$$result = x \left\{ \begin{array}{l} < \\ \leq \\ = \\ \geq \\ > \end{array} \right\} y ;$$

The operands can have any arithmetic types. The comparison is performed in their common type. The result type is BOOLEAN.

10.5.8 Absolute Value Built-in Function

$$result = ABS (x) ;$$

The result is the absolute value of x .

10.5.9 Integer Exponentiation Operator

$$result = x * * y ;$$

The types of the operands must be integer types. The exponent, y , must be a constant-expression whose value ≥ 0 , and x^y must not exceed the maximum value in x 's base ordinal type (which is INTEGER or LARGE_INTEGER).

10.5.10 SIGN Built-in Function

result = SIGN (x) ;

The argument x can have any arithmetic type. The type of the result is INTEGER; the value is 1 if $x > 0$, 0 if $x = 0$, and -1 if $x < 0$.

10.5.11 ODD Built-in Function

result = ODD (x) ;

The argument x must have an integer type. The type of the result is BOOLEAN; the value is TRUE if $x \text{ MOD } 2 == 1$, FALSE otherwise.

10.5.12 MAX Built-in Function

result = MAX (x₁, x₂ [, x₃...]) ;

The arithmetic MAX function accepts two or more arguments. The result's value is the value of the maximum argument, and the result's type is the common arithmetic type of the arguments.

10.5.13 MIN Built-in Function

result = MIN (x₁, x₂ [, x₃...]) ;

The arithmetic MIN function accepts two or more arguments. The result's value is the value of the minimum argument, and the result's type is the common arithmetic type of the arguments.

10.6 Boolean Operations

For all the operations in this section, the operands must have a base ordinal type of BOOLEAN, and the result type is BOOLEAN.

10.6.1 Boolean Complement Operator

result = NOT a ;

If a is FALSE, the result of this operation is TRUE; otherwise, the result is FALSE.

10.6.2 AND Operator

result = a AND b ;

If both a and b are TRUE, the result of this operation is TRUE; otherwise, the result is false.

This operation is conditional. If the value of operand a is FALSE, then operand b will not be evaluated in any way that can cause an exception or invoke a function with side effects; for example:

```
IF p <> NIL AND p^.rank > 0 THEN ...
```

If the value of p is NIL, it will not be used as an address to access the field $p^.rank$.

10.6.3 OR Operator

```
result = a OR b ;
```

If both a and b are FALSE, the result of this operation is FALSE; otherwise, the result is TRUE.

This operation is conditional. If the value of operand a is TRUE, then operand b will not be evaluated in any way that can cause an exception or invoke a function with side effects; for example, this code will not cause a zero-divide exception:

```
IF x == 0 OR y/x > 1 THEN ...
```

10.6.4 Boolean Comparison Operators

$$\text{result} = a \left\{ \begin{array}{l} < \\ <= \\ == \\ <> \\ >= \\ > \end{array} \right\} b ;$$

The ordinal values of a and b are compared. Note that, for this comparison, FALSE < TRUE.

10.6.5 Boolean Exclusive OR Operator

```
result = a XOR b ;
```

If a is FALSE and b is TRUE, or if a is TRUE and b is FALSE, the result of the operation is TRUE; otherwise, it is FALSE.

10.7 Ordinal Operations

10.7.1 Ordinal MAX Built-in Function

```
result = MAX (x1, x2 [, x3...]) ;
```

The ordinal MAX function accepts two or more arguments. They must have the same base ordinal types, but not INTEGER or LARGE_INTEGER (these are handled by the arithmetic MAX). The result's value is the value of the maximum argument, and the result's type is the base ordinal type of the arguments.

10.7.2 Ordinal MIN Built-in Function

```
result = MIN (x1, x2 [, x3...]) ;
```

The ordinal MIN function accepts two or more arguments. They must have the same base ordinal types, but not INTEGER or LARGE_INTEGER (these are handled by the arithmetic MIN). The result's value is the value of the minimum argument, and the result's type is the base ordinal type of the arguments.

10.7.3 Ordinal Comparison Operators

$$\text{result} = a \left\{ \begin{array}{l} < \\ \leq \\ = \\ < > \\ \geq \\ > \end{array} \right\} b ;$$

The ordinal values of a and b are compared; they must have the same base ordinal type.

10.8 Set Operations

The operands of set operators must have set types with the same range. (Requiring that the set ranges be identical eliminates situations that produce very complex code; the CONVERT_SET function [see Section 10.11.5.4] can be used to explicitly adjust ranges.) Unless specified otherwise, the result's type is a set type with the same range as the operands.

10.8.1 Set Complement Operator

$$\text{result} = - x ;$$

The result contains exactly those elements that are not members of x .

10.8.2 Set Union Operator

$$\text{result} = x + y ;$$

The result contains every element that is a member of x or y or both.

10.8.3 Set Intersection Operator

$$\text{result} = x * y ;$$

The result contains every element that is a member of both x and y .

10.8.4 Set Difference Operator

$$\text{result} = x - y ;$$

The result contains every element that is a member of x but not a member of y .

10.8.5 Set Exclusive OR Operator

result = x (+) y ;

The result contains every element that is a member of x or y , but not both.

10.8.6 Set Comparison Operators

$$\text{result} = x \left\{ \begin{array}{l} < \\ <= \\ == \\ <> \\ >= \\ > \end{array} \right\} y ;$$

The result is **BOOLEAN**. Sets are compared by set inclusion; $x < y$ is **TRUE** if every member of x is a member of y , and y has at least one member that is not a member of x .

The set comparison operators produce a result of **TRUE** in the following cases, and **FALSE** otherwise:

- $x == y$ Every member of x is a member of y and every member of y is a member of x .
- $x <> y$ There exists a member of x that is not a member of y , or there exists a member of y that is not a member of x .
- $x <= y$ Every member of x is a member of y .
- $x < y$ Every member of x is a member of y , and y has at least one member that is not a member of x .
- $x >= y$ Every member of y is a member of x .
- $x > y$ Every member of y is a member of x , and x has at least one member that is not a member of y .

10.9 Character String Operations

Except as otherwise specified, the operands of character string operators must have the type **STRING** or **CHARACTER**, and the result's type is **STRING**. (An operand of type **CHARACTER** is treated as being **STRING(1)**.) Note that when a data-reference, S , of type **VARYING_STRING** is interpreted as a value (as happens for operands), the value's type is **STRING(n)**, where n is the current value of $S.LENGTH$.

10.9.1 String Concatenation Operator

result = x + y ;

The result's type is **STRING($m + n$)**, where m is the length of x , and n is the length of y . The result is the characters of x followed by the characters of y :

$$\begin{aligned} \text{result}[i] &= x[i], \text{ for } i = 1..m \\ \text{result}[m + i] &= y[i], \text{ for } i = 1..n \end{aligned}$$

10.9.2 FIND_MEMBER Built-in Function

The FIND_MEMBER function is used to find the position, in a string, of the first character in a specified set of characters.

```
result = FIND_MEMBER (s, charset [, start]) ;
```

Arguments

s. This argument is a string that is searched for characters contained in *charset*.

charset. This argument must be of a set type whose base ordinal type is CHARACTER.

start. This optional argument is an integer; if present, its value must be > 0 and $\leq s.LENGTH$. If *start* is omitted, the value one is used. The *start* argument gives the character position in the string at which the search begins.

Result

The result is the minimum positive integer *k* such that $s[start + k - 1]$ is a member of *charset*, and $s[start + i - 1]$ is *not* a member of *charset*, for all $i < k$.

If *s* is the null string, or if no characters of $s[start..]$ are in *charset*, the result is zero.

10.9.3 FIND_NONMEMBER Built-in Function

The FIND_NONMEMBER function is used to find the position, in a string, of the first character that is not in a specified set of characters.

```
result = FIND_NONMEMBER (s, charset [, start]) ;
```

Arguments

s. This argument is a string that is searched for characters contained in *charset*.

charset. This argument must be of a set type whose base ordinal type is CHARACTER.

Result

start. This optional argument is an integer; if present, its value must be > 0 and $< s.LENGTH$. If *start* is omitted, the value one is used. The *start* argument gives the character position in the string at which the search begins.

The result is the minimum positive integer *k* such that $s[start + k - 1]$ is not a member of *charset*, and $s[start + i - 1]$ is a member of *charset*, for all $i < k$.

If *s* is the null string, or if all characters of $s[start..]$ are in *charset*, the result is zero.

10.9.4 FIND_SUBSTRING Built-in Function

```
result = FIND_SUBSTRING (s, pattern) ;
```

Arguments

s. This argument is a string that is searched for the string contained in *pattern*.

pattern. This argument is the string to be searched for.

Result

The result is an INTEGER that is the index of the first occurrence of *pattern* in *s*; that is, if *k* is the result, and *m* is the length of *pattern*, then: $s[k+i-1] = \text{pattern}[i]$, for all $i = 1..m$.

The relationship above is false for any smaller value of *k*.

The result is zero if *pattern* has nonzero length and does not occur in *s*, or if *pattern* is longer than *s*. The result is 1 if *pattern* is the null string, regardless of *s*'s value.

10.9.5 TRANSLATE_STRING Function

```
result = TRANSLATE_STRING (string, new [,OLDCHARS = old]) ;
```

Arguments

string. This argument is a string to translate.

new. This argument is a string that supplies the characters to swap in.

old. This optional argument is a string that dictates which characters in *string* are converted to the characters in *new*. If *old* is present, *new* and *old* must be character-string-literals or named constants of the same length, with none of *old*'s characters appearing more than once in *old*.

Note that this argument can only be supplied using keyword notation.

Result

The TRANSLATE_STRING function converts certain characters in *string*. Together, *new* and *old* determine a string that is used as a translation table to create the result string. The translation table is constructed as follows:

- If *old* is omitted, *new* is used as the translation table.
- If *old* is present, the compiler constructs a 256 character translation table that contains a translation for each character, *c*, in the character set. If *c* appears in *old*, then the translation for *c* is the corresponding character in *new*. If *c* does not appear in *old*, then *c* is its own translation.

The compiler constructs a translation table for all 256 characters. The table translates each character in *old* into the character in the corresponding position in *new* (other characters remain unchanged); for example:

```
r = TRANSLATE_STRING (S, "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                     OLDCHARS = "abcdefghijklmnopqrstuvwxyz");
```

The variable *r* will contain the letters of *S* translated to uppercase.

The result is a STRING(*n*), where *n* is the number of characters in *string*. The result string is obtained by taking the integer value of a character in *string*, and using that as an index into the translation table, *t*. That is, for *i* from 1 to *n*:

$$\text{result}[i] = t(\text{ORD}(s[i]))$$

A range violation occurs if the index, ORD(*s*[*i*]), exceeds the length of the translation table or if *new*'s length exceeds 256. (Note that the first kind of range violation is possible only if *old* is not specified.)

10.9.6 String Comparison Operators

$$\text{result} = x \left\{ \begin{array}{l} < \\ \leq \\ = \\ <> \\ \geq \\ > \end{array} \right\} y ;$$

The operands are compared lexically, based on the characters' ordinal values. If one operand is shorter, it is considered to be extended by blanks for comparison. If x and y have equal length, and contain the characters $x_1..x_n$ and $y_1..y_n$, then $x < y$ only if for some m in $1..n$, $\text{ORD}(x_m) < \text{ORD}(y_m)$ and $x_k = y_k$, for all $k < m$. The result's type is **BOOLEAN**.

10.10 Pointer Operations

10.10.1 Pointer Addition Operator

$$\text{result} = p + n ;$$

The operand p must have a pointer type. The operand n must have the type **INTEGER**.

The result is the sum of p and n , where both are treated as integers according to the system's treatment of addresses. The result's type is that of p . A range violation occurs if the result lies outside the range of addresses (if, for example, an address wraps around from a high value to a low value).

10.10.2 Pointer Subtraction Operator

$$\text{result} = p - x ;$$

The operand p must have a pointer type. The operand x must have a type that is equivalent to the type of p , or have the type **INTEGER**.

If p is a pointer and x is an integer, the result's type is that of p . The result is the difference $p - x$, where p and the result are treated as integers according to the system's treatment of addresses. A range violation occurs if the result lies outside the range of addresses (if, for example, an address wraps around from a high value to a low value).

If p and x are both pointers, the result is an **INTEGER**. The result is the difference $p - x$, where p and x are treated as integers according to the system's treatment of addresses. A range violation occurs if the result lies outside the range of **INTEGER**.

10.10.3 ADDRESS Built-in Function

$$\text{result} = \text{ADDRESS}(\text{reference}) ;$$

Argument

reference. This argument is interpreted as a data-reference to obtain a location *loc* with type *t*, which must not be a bit-class type.

Result

The result is *loc*, with type *POINTER t*.

If *reference* denotes a non-BIND parameter of procedure *p*, the address obtained from the ADDRESS function must not be used after the current invocation of *p* terminates. Violation of this rule is an error with unpredictable consequences.

10.10.4 CONTAINING_RECORD Built-In Function

```
result = CONTAINING_RECORD (field_address, record_type, field_name) ;
```

Arguments

field_address. This argument is a pointer value and it must be the address of the specified field within an existing instance of *record_type* (if not, the consequences are unpredictable). The type of *field_address* must be *POINTER t*, where *t* is the type of the named field.

record_type. This argument must be a name denoting a record type.

field_name. This argument must be the name (unqualified) of a field in *record_type*. The field must have a constant offset.

Result

The result is the address of the record denoted by *record_type*; its type is *POINTER record_type*.

10.10.5 Pointer Comparison Operations

$$\text{result} = p \left\{ \begin{array}{l} < \\ <= \\ == \\ <> \\ >= \\ > \end{array} \right\} q ;$$

The operands *p* and *q*, which must have equivalent pointer types, are interpreted as integers according to the system's treatment of addresses. The result's type is *BOOLEAN*.

NOTE

Except for the operators “==” and “<>,” the result is system dependent; the use of one of the other operators could produce different results in separate runs of a program, even on the same system with the same input data.¹

Also, a comparison between pointers that address parts of the same record (or other allocation unit) are unreliable because distinct fields can have the same address if one field is zero-sized.

¹ Should Pillar even allow comparison operators other than == and <> on pointers?

10.11 Operations Related to Types

10.11.1 MAX Built-In Function

result = MAX (tname) ;

The argument *tname* must be a name denoting an ordinal type, *t*. The result is the highest value in *t*'s range, and has the type of *t*'s base ordinal type.

10.11.2 MIN Built-In Function

result = MIN (tname) ;

The argument *tname* must be a name denoting an ordinal type, *t*. The result is the lowest value in *t*'s range, and has the type of *t*'s base ordinal type.

10.11.3 DATA_TYPE_SIZE Built-In Function

The DATA_TYPE_SIZE built-in function provides the size of an item.

result = DATA_TYPE_SIZE (units, item [, extent-value-list]) ;

Arguments

units. This argument must be one of the names BIT, BYTE, WORD, LONGWORD, or QUADWORD.

item. This argument must be a named-type or a data-reference.

extent-value-list. This optional argument is a list of one or more expressions. If *item* is not the name of a flexible type, then *extent-value-list* must not be present. If *item* is the name of a flexible type, then *extent-value-list* must contain exactly as many arguments as there are extents in *item*. For example, the value of DATA_TYPE_SIZE (BYTE, STRING, 10) is 10.

Result

The result is the size of *item*'s type in the specified units. The specified units must not exceed the size units of *item*'s type as determined by Pillar's data representation rules; for example, if *item*'s type is LONGWORD_DATA, *units* can be BYTE, but not QUADWORD.

10.11.4 FIELD_OFFSET Built-In Function

The FIELD_OFFSET built-in function provides the magnitude of an item's offset.

result = FIELD_OFFSET (units, item, field_name [,extent-value-list]) ;

Arguments

units. This argument must be one of the names BIT, BYTE, WORD, LONGWORD, or QUADWORD.

item. This argument must be a named-type denoting a record type, or a data-reference whose type is a record type.

field_name. This argument must be the name of a field in the record type.

extent-value-list. This optional argument is a list of one or more expressions. If *item* is not the name of a flexible type, then *extent-value-list* must not be present. If *item* is the name of a flexible type, then *extent-value-list* must contain exactly as many arguments as there are extents in *item*.

Result

The result is the offset of *item* from the record origin in the units specified by the argument *units*. The specified units must not exceed the units of *field_name*'s offset in *item*. (This offset is determined by the layout rules if *item* is specified with a layout option; otherwise, it is determined by the compiler.)

10.11.5 Conversion Functions

The built-in conversion functions provide controlled conversions between data types.

$$\text{result} = \left\{ \begin{array}{l} \text{CONVERT_ORDINAL} \\ \text{CONVERT_POINTER} \\ \text{CONVERT_ARITHMETIC} \\ \text{CONVERT_SET} \\ \text{CONVERT_STRING} \\ \text{CONVERT_UNTYPED} \end{array} \right\} (\text{source_value} [, \text{option...}] [, \text{TARGET} = \text{target_type}]) ;$$

Arguments

source_value. This argument is the expression whose value is to be converted.

option. These optional arguments are names, and vary among conversion functions. An option cannot be specified more than once. A specific option is allowed only in the cases indicated in the definitions of the individual conversion functions.

target_type. This optional argument is a named-type specifying the target type for the conversion. If it is not present, context must provide the target type for the conversion.

Result

The result is the source value converted to the target type according to the rules for the specified function.

10.11.5.1 CONVERT_ORDINAL Built-In Function

Source_value must have an ordinal type, and the target type must be an ordinal type. No options are allowed.

The result's ordinal value is the ordinal value of *source_value*, and the result's type is that of the target type. A range violation occurs if the result's value lies outside the target type's range.

10.11.5.2 CONVERT_POINTER Built-In Function

Source_value must have a pointer type with associated type *at*, and the target type must be a pointer type with associated type *at1*.

The options allowed are IGNORE_ALIGNMENT and IGNORE_SIZE.

This function allows the conversion between pointers with different associated types without changing the representation of the source. The pointer conversion alignment rule requires that one of the following must hold:

- The alignment requirements of *at* and *at1* are the same.
- Either *at* or *at1* is ANYTYPE.
- The IGNORE_ALIGNMENT option is specified.

The pointer conversion size rule requires that one of the following must hold:

- The sizes of *at* and *at1* are equal.
- Either *at* or *at1* is ANYTYPE.
- *At* and *at1* are both record types, and one extends the other.
- The IGNORE_SIZE option is specified.

10.11.5.3 CONVERT_ARITHMETIC Built-in Function

Source_value must have an arithmetic type, and the target type must be an arithmetic type.

The mutually exclusive options ROUND and TRUNCATE can be specified. If either is specified, the source or target or both must be floating point.

If no option is specified, the normal conversion rules apply, as defined in Chapter 5.

If ROUND or TRUNCATE is specified and an actual value conversion is necessary, then rounding or truncation is used instead of the default.

10.11.5.4 CONVERT_SET Built-in Function

Source_value must have a set type, and the target type must be a set type.

This function allows conversion between set types with different ranges. The target and source types must have equivalent base ordinal types, but their ranges can differ.

The only option allowed is TRUNCATE. If TRUNCATE is specified, the source value can have elements outside the target type's range, and they are simply ignored. If TRUNCATE is not specified, then all the elements in the source value must lie in the target type's range; otherwise, a range violation occurs.

10.11.5.5 CONVERT_STRING Built-in Function

Source_value must have a string type, and the target type must be a string type.

The only option allowed is TRUNCATE. A range violation occurs if the TRUNCATE option is not specified and the length of *source_value* is greater than the length of the target type. If the TRUNCATE option is specified, *source_value* is truncated to the length of the target type if necessary.

VARYING_STRING cannot be explicitly specified as the target type.

10.11.5.6 CONVERT_UNTYPED Built-In Function

This function provides the ability to retype a value without changing its internal representation except for an adjustment in size.

The options allowed are TRUNCATE, ZERO_EXTEND, and SIGN_EXTEND. No more than one option can be specified. If no option is specified, a range violation occurs if *source_value* and the target type have different sizes.

If TRUNCATE is specified, only the first n bits of *source_value* are used, where n is the size in bits of the target type. A range violation occurs if the size of *source_value* is greater than the size of the target type.

If ZERO_EXTEND or SIGN_EXTEND is specified, *source_value*'s internal representation is sign- or zero-extended to the size of the target type. This happens regardless of what *source_value*'s internal representation is; *source_value* might or might not be data that contains a sign bit. A range violation occurs if the size of *source_value* is greater than the size of the target type.

The target type of CONVERT_UNTYPED cannot be VARYING_STRING (neither implicitly nor specified explicitly). There are no other type restrictions on the target type or *source_value*, since the CONVERT_UNTYPED function is concerned only with the size of the target and source.

10.11.6 INITIALIZE_FIELDS Built-In Function

INITIALIZE_FIELDS (pointer, [, extent-value-list]);

The INITIALIZE_FIELDS function allows the initialization of record fields declared with an initializer and captured extents.

Arguments

pointer. This argument supplies a value of type POINTER t , where t is a record type that is declared as having captured extents or as having an initializer on at least one field.

extent-value-list. This optional argument is a list of one or more expressions. It must be present if and only if the type t (defined above) is a type with captured extents. If present, extent-value-list must contain exactly as many arguments as there are captured extents in t .

INITIALIZE_FIELDS initializes the following fields in the record addressed by *pointer*:

- Each argument in *extent-value-list* is assigned (in the order that the captured extents appear in t 's declaration) to one of the record's captured extents.²
- Each of the fields that is specified in t 's declaration with an initializer is assigned the value specified in that initializer.

² Should Pillar allow the extent-value-list in INITIALIZE_FIELDS, DATA_TYPE_SIZE, and FIELD_OFFSET to be specified using keyword notation?

10.12 Miscellaneous Built-in Functions

10.12.1 ZERO Built-in Function

```
result = ZERO ( [ TARGET = target_type ] );
```

Arguments

target_type. This optional argument is a named-type specifying a target type. If it is not present, context must provide a target type.

Result

The result's type is that of the target type. The result has all bits zero, except for any captured extents present in the target type.

10.12.2 ARGUMENT_PRESENT Built-in Function

```
result = ARGUMENT_PRESENT (parameter) ;
```

Argument

parameter. This argument is a name denoting an OPTIONAL parameter of a procedure containing this invocation of ARGUMENT_PRESENT.

Result

The result type is BOOLEAN. The result value is TRUE if an argument was passed to the parameter in the current invocation of the procedure, FALSE otherwise.

10.12.3 VALIDATE_VALUE Built-in Function

The VALIDATE_VALUE built-in function is used to ensure that a value is in the range required by its type definition. This value is always an ordinal value in Pillar.

This function is especially useful in system services because many of their input values (direct and indirect through data structures) are ordinal values: buffer lengths, string lengths, counts, and various codes. The system service must check that the actual value it receives is within the range of the item's declared ordinal type. If it does not check, and such a value is out of range when input, the service can fail in an unpredictable manner. The programmer of a system service *must not make any assumptions about the effects of a range discrepancy*. This range problem is not peculiar to system services, however; it occurs whenever a programmer constructs a routine to be called from other languages that are invulnerable to invalid arguments.

```
result = VALIDATE_VALUE (reference [, type-name]) ;
```

Arguments

reference. This argument is a data-reference whose type is an ordinal type.

type-name. This optional argument is the name of an ordinal type, *t*. If *type-name* is omitted, *t* is taken to be the exact ordinal type of *reference*.

Result

The function compares the actual integer value of *reference* with the range of integer values defined by the ordinal type *t*, and returns a BOOLEAN result: TRUE if the value is in range, FALSE otherwise. For example:

```
PROCEDURE p( IN s : string(*) RETURNS STATUS;
BEGIN
  IF NOT validate_value(s.length) THEN
    RETURN some_status_code;
  END IF;
```

10.12.4 VALIDATE_ALIGNMENT Built-in Function

The VALIDATE_ALIGNMENT built-in function is used to verify that a data item has the correct alignment.

The Pillar language defines an alignment requirement for each data type, and the compiler assumes that a data item satisfies the alignment requirement of its type, unless the item is a record field that is explicitly dealigned by use of a LAYOUT option. If the actual alignment of a data item is less than the compiler's assumed alignment, accessing the item can cause an alignment fault on some systems.

For some routines, such as system services, the requirement of correct data alignment should be enforced. Other routines might also want to check for unaligned input data and take special action.

```
result = VALIDATE_ALIGNMENT (reference [, type-name]) ;
```

Arguments

reference. This argument is a data-reference to an item whose alignment is to be validated. This argument must be addressable; it cannot have a bit-class type or a pointer type.³

type-name. This optional argument is the name of a type. This argument does not have to be the type of *reference*; it can be any type except for bit-class data types and ANYTYPE. If *type-name* is omitted, the type is taken to be the type of *reference*. The *type-name* argument cannot be omitted if *reference*'s type is ANYTYPE.

Result

The function compares the actual alignment of *reference* with the the alignment requirement of the type specified by *type-name*, and returns a BOOLEAN type result; the result value is TRUE if the alignment is at least that required, FALSE otherwise. For example:

```
PROCEDURE p( IN Q : POINTER quadword_data(*) RETURNS STATUS;
BEGIN
  IF Q==NIL THEN
    NOTHING;
  ELSEIF NOT validate_value(Q.length) THEN
    RETURN some_status_code;
  ELSEIF NOT validate_alignment(Q^) THEN
    RETURN some_status_code;
  END IF;
```

³ Pointer types are prohibited in order to detect the error of specifying "p" rather than "p^" as *reference*.

As stated above, *reference* cannot be a pointer type. To check the alignment of a *pointer* (as opposed to the data it addresses, which is a more usual case), type casting can be used; for example, the following call validates the pointer *p* as being *longword* aligned:

```
result = VALIDATE_ALIGNMENT(p::{ANYTYPE}, longword);
```

CHAPTER 11

STATEMENTS

This chapter describes all of Pillar's statements, including compound statements and statement-sequences.

11.1 Control Flow and Statement Sequences

To explain the effects of nonsequential control flow, execution of an individual statement is said to terminate *sequentially*, with an *exit-loop*, with a *procedure return*, with a *GOTO*, or with an *exception*. In a set of nested constructions, a nonsequential termination ripples up to the construction containing its target; for example:

```
LOOP
  .
  .
  .
  BEGIN
  .
  .
  .
  IF ... THEN EXIT LOOP; END IF;
  END
  .
  .
  .
  ! more of the loop
END LOOP;
```

The statement `EXIT LOOP;` is said to terminate with an exit-loop. The containing `if`-statement and the main statement-sequence of the `BEGIN-END` local-block terminate in the same way. At this point we have reached the loop-statement to which the exit-loop-statement applies, and the loop-statement simply terminates sequentially.

Of course, the code does not really do all this. Normally (if there is no unwind handler as described in Section 12.6 to invoke), the `EXIT LOOP` will simply generate a branch instruction to the code following the loop.

In the description of individual statements, nonsequential termination is discussed only when it is directly related to the particular statement. For a compound-statement (that is, one that contains a statement-sequence), nonsequential termination of a contained statement-sequence terminates the compound-statement in the same way, unless stated otherwise. Expression evaluation can cause a statement to abruptly terminate with an exception or nonlocal `GOTO`. Also, range violations in a statement's own semantics cause it to terminate with an exception, if range checking is enabled.

Sequential control flow occurs within a statement-sequence, which is the only syntactic category that refers to the statement category.

■ statement-sequence

{ [identifier :] statement } ...

The identifiers are declared as names of labels. The scope of the declaration is the statement-sequence.

The statements are executed in order until the last statement has been executed, a condition occurs, or a statement terminates nonsequentially. If the last statement is executed and terminates sequentially, the entire statement-sequence terminates sequentially.

If a statement terminates with a GOTO to a label declared in the same statement-sequence, execution of the statement-sequence continues from the label. In all other cases of nonsequential termination of a contained statement, the statement-sequence terminates in the same manner as the terminating statement.

If a condition occurs, execution of the statement-sequence is suspended and a condition handler is found. Depending on the handler, one of the following will occur:

- The suspended statement continues after the handler's execution terminates.
- The suspended statement terminates with a nonlocal GOTO executed by the handler (or a routine called by the handler).
- The suspended statement terminates with an exception, then the exception handler is invoked; this is the normal case in Pillar.

Condition and exception handling is discussed further in <REFERENCE>(Local_Blocks_Ex_Hand_SECTION).

■ statement

{
 assert-statement
 assignment-statement
 built-in-function-call-statement
 case-statement
 compound-statement
 exit-loop-statement
 goto-statement
 if-statement
 loop-statement
 nothing-statement
 procedure-call-statement
 raise-statement
 return-statement
}

The interpretation of each statement category is described in its own section in this chapter.

11.2 ASSERT Statement

An assert-statement is used to make assertions. Violations of the assertions are treated in the same manner as range violations.

■ assert-statement

```
ASSERT { expression [ ELSE character-string-literal ] } , ... ;
```

The expression's type must be **BOOLEAN**. To execute an assert-statement, each assertion expression is evaluated. If any assertion's value is **FALSE**, an exception occurs. Note that if more than one of the assertions is false, any one of the false assertions can be picked as the failing assertion.

Code will not be generated for an assert-statement unless assertion checking is enabled when the module is compiled. However, if the expression is constant and **FALSE**, it will always be detected by the compiler and treated as a range violation involving constant expressions. Also, the compiler can treat the assertion as true, for code optimization purposes, even if assertion checking is disabled.

If the assertion is of the form *expression ELSE "character-string-literal"*, the string literal is used as the substitution argument for the Pillar condition that is raised. The condition name **PILLAR\$_ASSERT** is defined by the compiler as though it had the following declaration:

```
CONDITION  
PILLAR$_ASSERT : ERROR = "!";
```

Note that there is also a predeclared **BOOLEAN** constant, **ASSERT_CHECK_ENABLED**, whose value is **TRUE** if a module is compiled in a mode that guarantees the checking of assert-statements. **ASSERT_CHECK_ENABLED** can be used, in conjunction with an assert-statement or **RAISE ERROR**, to control checking of conditions too complicated to be specified by a simple assert-statement.

11.3 ASSIGNMENT Statement

An assignment-statement is used to assign a value to a variable.

■ assignment-statement

```
data-reference = expression ;
```

The data-reference is interpreted to yield a location that must be *assignable* (see Chapter 9). The location has a type *t*, which is used as the target type while evaluating the expression. Evaluating the expression can cause the expression's value to be converted to type *t*, as described in Section 5.16.4. The value (possibly converted) of the expression is assigned to the location.

11.4 CASE Statement

A case-statement is used to select one of several statement-sequences for execution according to the value of an ordinal expression. The use of a case-statement suggests the use of a jump table of some sort. The compiler will generate a different style of code for some special cases, but usually it uses a jump table.

■ case-statement

```
CASE expression  
{ WHEN set-of-values THEN statement-sequence } ...  
[ WHEN OTHERS THEN statement-sequence ]  
END CASE ;
```

The selector expression after CASE must have an ordinal type. Each "WHEN set-of-values THEN" specifies a selector set for the statement-sequence following the THEN. All expressions and ranges used in the set-of-values must be constant, and all must have base ordinal types that are equivalent to the base ordinal type of the selector expression. There is no target type for interpretation of the set-of-values.

The selector sets must be disjoint so that a particular ordinal value will select only one of the contained statement-sequences; a given ordinal value cannot occur or be contained in more than one selector.

To execute a case-statement, the selector expression is evaluated. If the resulting value is in one of the selector sets, the associated statement-sequence is executed; otherwise, there must be a "WHEN OTHERS THEN" (if not, range violation), and the statement-sequence following it is executed.

11.5 Compound Statement

A compound-statement is used to establish a scope for declarations and exception handling. However, compound-statements do not have a stack frame or any similar run-time structure.

■ compound-statement

[WITH] local-block ;

WITH is used only if the local-block contains a local-block-declaration-section or an enable-section.

To execute a compound-statement, the local-block is executed.

11.6 EXIT LOOP Statement

The exit-loop-statement is used to terminate the execution of the innermost containing loop-statement.

■ exit-loop-statement

EXIT LOOP ;

The exit-loop-statement must be contained in a loop-statement, but the containment can be through one or more levels of statement-sequence nesting.

The exit-loop statement terminates with an exit-loop.

11.7 GOTO Statement

A goto-statement is used to transfer control to a labeled point in a statement-sequence.

■ goto-statement

GOTO identifier ;

The identifier is interpreted as a reference. The reference must denote a label. The goto-statement terminates with a GOTO to the specified label.

Pillar's scope rules ensure that the target label belongs either to the current statement-sequence, or to one at a higher level. A jump to a label in a statement-sequence can occur only when control is already within the statement-sequence.

A GOTO can jump out of the current procedure. However, such a nonlocal GOTO can only jump to a label in the main statement-sequence of a local-block containing the current procedure (at some level of nesting).

11.8 IF Statement

An if-statement is used to conditionally execute a statement-sequence. The syntax provides a convenient form for sequential selection using **BOOLEAN** test expressions.

■ if-statement

```
IF expression THEN statement-sequence
  [ ELSEIF expression THEN statement-sequence ] ...
  [ ELSE statement-sequence ]
END IF ;
```

The test expressions after **IF** and **ELSEIF** must have type **BOOLEAN**.

When an if-statement executes, the test expressions are evaluated in order until one yields the result **TRUE**, or all have been evaluated. Tests expressions following the one yielding **TRUE** are not evaluated.

If some test expression yields **TRUE**, the statement-sequence introduced by the following **THEN** is executed. Otherwise, if the if-statement contains **ELSE**, the statement-sequence following the **ELSE** is executed.

11.9 LOOP Statement

A loop-statement is used to execute a statement-sequence zero or more times. Termination can be controlled by an explicit loop-control or **WHILE expression**, or the loop can be indefinite. In either case, nonsequential termination of the statement-sequence terminates the loop. The simplest form of nonsequential termination is through an exit-loop-statement, which terminates the innermost containing loop.

■ loop-statement

```
[[ ordinal-type-control
  increment-or-decrement-control
  general-control ]] [ WHILE expression ]
LOOP
statement-sequence
END LOOP [ identifier ] ;
```

If a loop contains an ordinal-type-control, increment-or-decrement-control, or general-control, then the identifier following **END LOOP** must be the name of the loop's control variable; otherwise, the identifier must not be present.

If the loop-statement contains a **WHILE**, the expression after **WHILE** must have type **BOOLEAN**.

To execute a loop-statement, the following steps are performed:

1. If the loop-statement contains an ordinal-type-control, increment-or-decrement-control, or general-control, the control is initially evaluated as described under its syntax heading. If that evaluation specifies loop termination, the execution of the loop statement terminates sequentially, and the statement-sequence is not executed.
2. If the loop-statement contains a WHILE, the expression after the WHILE is evaluated. If the result is FALSE, execution of the loop-statement terminates sequentially; otherwise, execution proceeds with step 3.
3. The statement-sequence is executed. If it terminates sequentially, execution of the loop proceeds with step 4; otherwise, execution terminates.

If the statement-sequence terminates with an exit-loop, execution of the loop-statement terminates sequentially. If the statement-sequence terminates with a procedure return, GOTO, or exception, the loop-statement terminates in the same manner as the statement-sequence.

4. If the loop-statement does not contain a control, execution proceeds with step 2. If there is a control, it is evaluated for continuation as described under its syntax heading. If that evaluation specifies loop termination, the execution of the loop-statement terminates sequentially; otherwise execution proceeds with step 2.

11.9.1 Loop Control by an Ordinal Type

An ordinal-type-control is used to perform the body of a loop with a control variable taking on all values in an ordinal type's range. In this case, only the ordinal-type-control can assign a value to the control variable.

■ ordinal-type-control

FOR identifier IN named-type

The identifier is declared as the name of a control variable. The scope of this declaration is the entire loop-statement containing this control.

Initial Evaluation

Initial evaluation applies at the beginning of a loop (step 1 in Section 11.9). The named-type is interpreted; it must be the name of an ordinal type, and this is made the type of the control variable. The minimum value in the type's range is made the initial value of the control variable. The maximum ordinal value in this type's range must be less than the maximum value in the range of the type INTEGER (otherwise, range violation).

Evaluation for Continuation

Evaluation for continuation applies at the end of the loop (step 4 in Section 11.9). If the current value of the control variable is the maximum value in its type's range, the loop terminates; otherwise, the next value in the range is made the value of the control variable.

11.9.2 Loop Control by Increment or Decrement

An increment-or-decrement-control is used to perform the body of a loop with an ordinal control variable taking on equally spaced values. The direction in which the values change must be explicitly specified by use of TO or DOWN TO.

By default, the ordinal value of the control variable changes by one on each iteration of the loop. However, a different spacing can be specified using the keyword BY. The value of the spacing is always given as a positive integer.

Only the increment-or-decrement-control can assign a value to the control variable.

■ increment-or-decrement-control

```
FOR identifier [ : named-type ] = expression  
  [ BY expression ] [ DOWN ] TO expression
```

The identifier is declared as the name of a control variable. The scope of this declaration is the entire loop-statement containing this control. The name must not be referred to within the loop-control.

Initial Evaluation

Initial evaluation applies at the beginning of a loop (step 1 in Section 11.9).

If a named-type is given, it must be the name of an ordinal type that is the control variable's type. The initial expression (the expression after the "=") is evaluated with the control variable's type as a target type, and the resulting value is the initial value of the control variable.

If no named-type is given for the control variable, its type is INTEGER. The initial expression, which denotes the control variable's initial value, must also have the type INTEGER.

If BY is used, the expression after BY is evaluated to yield the increment value. The type of the expression must be INTEGER, and the increment value must be greater than zero (otherwise, range violation). If BY is not used, the increment value is one.

The expression after TO is evaluated to yield a limit value. The type of this expression must be compatible with the control variable's type, and its value must lie within the range of the control variable's type (otherwise, range violation).

The initial value of the control variable is compared with the limit value. If DOWN TO is used, and the initial value is less than the limit value, the loop terminates. If only TO is used, and the initial value is greater than the limit value, the loop terminates.

Evaluation for Continuation

Evaluation for continuation applies at the end of the loop (step 4 in Section 11.9).

A new ordinal value for the control variable is determined using the current value and the increment value; the increment value is determined during the initial evaluation of the increment-or-decrement-control. If DOWN TO is used, the increment-value is subtracted from the current value; otherwise, the increment-value is added to the current value. The new ordinal value must lie within the range of the control variable's type (otherwise, range violation).

The new value of the control variable is compared with the limit value. If DOWN TO is used, and the new value is less than the limit value, the loop terminates. If only TO is used, and the new value is greater than the limit value, the loop terminates.

11.9.3 General Loop Control

A general-control is used to perform the body of a loop with a control variable whose value—after the first iteration—is determined by repeatedly evaluating an expression. A general-control does not, by itself, terminate the loop.

In this case, only the general-control can assign a value to the control variable.

■ general-control

```
FOR identifier [ : named-type ] = expression  
NEXT expression
```

The identifier is declared as the name of a control variable. The scope of this declaration is the entire loop-statement containing this control. The control variable must not be referred to within the initial expression. However, it can be referred to in the expression following NEXT.

Initial Evaluation

Initial evaluation applies at the beginning of a loop (step 1 in Section 11.9).

If a named-type is given, it is interpreted to yield the type of the control variable. The initial expression (the expression after the “=”) is evaluated with the control variable’s type as a target type, and the resulting value is the initial value of the control variable.

If no named-type is given for the control variable, its type is INTEGER. The initial expression, which gives the initial value of the control variable, must also have the type INTEGER.

Evaluation for Continuation

Evaluation for continuation applies at the end of the loop (step 4 in Section 11.9).

The expression after NEXT is evaluated with the control variable’s type as its target type. The resulting value is made the current value of the control variable.

11.10 NOTHING Statement

■ nothing-statement

```
NOTHING ;
```

Execution of a NOTHING statement has no effect.

11.11 Built-in Function Call Statement

■ built-in-function-call-statement

```
built-in-function-invocation ;
```

If the built-in function is one that returns a result, the result’s type must be STATUS. In this case, an exception occurs if the returned status is not success.

The interpretation of a built-in-function-invocation is described separately for each built-in-function. The predeclared built-in-functions that do not return a result are described in the following subsections.

11.11.1 READ_REGISTER Built-in Function

The READ_REGISTER built-in function performs a reference that is guaranteed to occur in an atomic operation, and will not be removed or altered by any optimizations the compiler carries out. Such an operation is necessary when accessing a device register, for example (hence the name).

```
READ_REGISTER (target, source) ;
```

Arguments

target. This argument is interpreted as a variable data-reference.

source. This argument is interpreted as the value of a variable data-reference, but it cannot be an automatic variable. The *source* argument's type must be constant and compatible with the type of *target*. The rules for internal representation of data must assign BYTE, WORD, LONGWORD, or QUADWORD size units to *source*'s type, and the size in these units must equal one. The value's actual alignment in storage must be consistent with the size units (otherwise, range violation).

Result

The source value is obtained from storage in a single atomic operation. It is then assigned to *target*, and converted, if necessary, according to Pillar's compatibility rules (see Section 5.16).

Data items for which an atomic read can be performed are specific to the target system. (Some target systems might be able to support atomic read *up to* a quadword, others only up to a longword.)

11.11.2 WRITE_REGISTER Built-in Function

In contrast to READ_REGISTER, WRITE_REGISTER performs an atomic assignment to storage. WRITE_REGISTER also provides a convenient means of constructing a record value.

```
WRITE_REGISTER (target, source, field_name1 = value1, field_name2 = value2,...) ;
```

Arguments

target. This argument, which cannot be an automatic variable, is interpreted as a variable reference. The rules for internal representation of data must assign BYTE, WORD, LONGWORD, or QUADWORD size units to *source*'s type, and the size in these units must equal one. The target's actual alignment in storage must be consistent with the size units (otherwise, range violation).

source. This argument is mutually exclusive with the *field_name* arguments. Either, but not both, must be present. It is interpreted as a value reference, and its type must be compatible with *target*'s type; the value is converted to *target*'s type if necessary.

value_n. These arguments are mutually exclusive with *source*. If specified, *target*'s type must be a record type, and *field_name_n* must be the name of a field in that record.

When the arguments *field_name_n=value_n* are specified, a source value with *target's* type is built by initially assigning ZERO, and then assigning each *value_n* to the corresponding field. The type of *value_n* must be compatible with the field's type; the value is converted to the field type if necessary.

Result

The source value (converted or constructed, as required) is assigned to the target item in a single atomic operation. The class of data items for which such an atomic write can be performed is the same as for atomic reads (see Section 11.11.1).

As an example of using WRITE_REGISTER to construct a record value, suppose the following declarations exist:

```
TYPE
  control_status_register: RECORD
    unit_number: integer [0..255] size(byte);
    go: boolean;
    clock_enable: boolean;
    error_flag: boolean;
  LAYOUT
    unit_number;
    fill1: filler(bit,5);
    go;
    fill2: filler(bit,2);
    clock_enable;
    error_flag;
    fill3: filler(bit,50);
  END LAYOUT;
END RECORD;

VARIABLE
  csr: control_status_register;
```

Then the following call to WRITE_REGISTER can be used to assign to *csr* in an atomic operation:

```
WRITE_REGISTER (csr, go = true, unit_number = 5) ;
```

11.12 Procedure Call Statement

A procedure-call-statement is used to invoke a procedure that does not return a value or that returns a value of type STATUS. In the latter case, the returned status is automatically checked; if it is not success, an exception occurs.

■ procedure-call-statement

```
procedure-invocation ;
```

If the procedure is one that returns a result, the result's type must be STATUS.

The procedure-invocation is interpreted as described in Section 13.4. If the procedure is one that returns a result of type STATUS, and a nonsuccess status is returned, an exception occurs.

11.13 RAISE Statement

A raise-statement is used to raise an exception or report. There are several forms:

■ raise-statement

$$\text{RAISE } \left\{ \begin{array}{l} \text{ERROR [character-string-literal]} \\ \left\{ \begin{array}{l} \text{EXCEPTION} \\ \text{REPORT} \end{array} \right\} \text{ data-reference [argument] , ...} \\ \text{VECTOR data-reference} \end{array} \right\} ;$$

11.13.1 RAISE ERROR

A raise-statement of the form RAISE ERROR is used to cause an exception; this is the most memory-efficient method for causing a nonspecific error exception. RAISE ERROR is not intended for reporting specific errors to users.

If the raise error is of the form *RAISE ERROR "character-string-literal"* the string literal is used as the substitution argument for the Pillar condition that is raised. The condition name PILLAR\$_ERROR is defined by the compiler as though it had the following declaration:

```
CONDITION
PILLAR$_ERROR : ERROR = "!";
```

The other forms of the raise-statement are used to signal a specified condition, and an argument for the condition's associated message can be given.

11.13.2 RAISE EXCEPTION and RAISE REPORT

The form RAISE EXCEPTION is used to signal a condition that will be treated as an exception. This means that a condition handler for this condition is not allowed to return control to this statement (that is, to continue); the system or the Pillar run time prevents it. An exception caused by an instruction is treated the same way.

RAISE REPORT is similar to RAISE EXCEPTION, except that it signals a condition that need not be treated as an exception; a condition handler is allowed to return control to the raise-statement, in which case sequential statement execution continues.

The data-reference in RAISE REPORT or RAISE EXCEPTION must be either a name declared as a condition or a reference of type STATUS.

The number of arguments present in RAISE REPORT or RAISE EXCEPTION must be exactly the number of substitution parameters in the condition's message. Any arguments present must be string expressions.

Condition handling is discussed in <REFERENCE>(Local_Blocks_Ex_Hand_SECTION).

11.13.3 RAISE VECTOR

The RAISE VECTOR form of this statement is provided because the condition to be signaled might already be specified by the contents of a message vector (described in Section 12.2.3). In the RAISE VECTOR statement, data-reference is a reference of type \$condition.message_vector (the \$condition module is described in Section 12.2). A handler invoked for this signal gets a message vector with the same contents as data-reference. A procedural handler (see

Section 12.4 on OZIX/OSF gets data-reference itself without any copying. The message handle in data-reference should denote a condition, not just a message.

A range violation occurs if the severity in the message vector is severity.no_severity (see Section 12.2.1).

11.14 RETURN Statement

A return-statement is used to return control from the current procedure-invocation. If the procedure is one that returns a result, the returned value can be given in the return-statement or taken from the procedure's result variable.

■ return-statement

```
RETURN [ expression ] ;
```

The following rules cover the interaction of RETURN statements, result-variables, and the RETURNS option in procedure declarations:

- Executing a RETURN statement that does not contain an expression is treated the same as executing the END statement of a procedure (also called *sequential termination*).
- A procedure declared without a RETURNS option cannot contain a RETURN statement that contains an expression.
- A procedure declared with a RETURNS option can contain a RETURN statement that contains an expression.
- A procedure declared with a RETURNS option that does not have a result-variable must contain at least one RETURN statement that contains an expression.
- A procedure declared with a RETURNS option can contain a RETURN statement that does not contain an expression only if the procedure has a result variable.
- If a procedure declared with a RETURNS option executes a RETURN statement that does not contain an expression, then its result variable must have been assigned a value (otherwise, range violation). The Pillar compiler might not detect this violation. One implication of this rule is that a range violation always occurs if the END statement is executed in a procedure declared with a RETURNS option, if that procedure has no result variable.
- Taking the address of a result-variable or passing it to a BIND parameter is not permitted.

Execution of the return-statement terminates with a procedure return.

CHAPTER 12

LOCAL BLOCKS AND EXCEPTION HANDLING

12.1 Local Blocks

Local-blocks occur as procedure bodies and as compound statements.

■ local-block

```
[ { local-block-declaration-section } ... ]  
[ enable-section ]  
BEGIN  
statement-sequence  
[ exception-handler-section ] ...  
[ unwind-handler-section ]  
[ subprocedure-section ]  
END
```

The *statement-sequence* immediately contained in a local-block is the block's *main statement-sequence*.

12.1.1 Declarations in a Local Block

■ local-block-declaration-section

```
{  
normal-type-declaration  
flexible-type-declaration  
complete-opaque-type-declaration  
procedure-type-declaration  
constant-declaration  
complete-value-declaration  
complete-variable-declaration  
bind-declaration  
define-declaration  
complete-message-declaration  
complete-condition-declaration  
}
```

12.1.2 Subprocedures

■ subprocedure-section

SUBPROCEDURES { complete-procedure-declaration } ...

A subprocedure-section is only allowed in the local-block of a procedure.

A procedure declared in the subprocedure-section of a procedure *p* is a *subprocedure* of *p*. A subprocedure can contain a subprocedure-section.

The name of each procedure declared in a subprocedure-section has a scope that is the same as the local-block in which it is declared.

12.1.3 Termination of a Statement Sequence

A *termination state* is *continue*, *unwind*, or an exception.

Execution of a statement-sequence *s* within a local-block *b* produces a termination state *t* relative to *b* by the following rules:

- If *s* terminates with an exception *e*, *t* is *e*.
- If *s* terminates with an explicit or implicit unwind of *b*, *t* is *unwind*.
- Otherwise, *t* is *continue*.

12.1.3.1 Explicit Unwinding

A statement-sequence *s* within local-block *b* terminates with an *explicit unwind* of *b* if a statement within *s* explicitly transfers control to a point outside *b*. More concretely, *s* explicitly unwinds *b* at the execution of:

- A return-statement, unless *b* is a procedure body
- An exit-loop-statement, if *b* occurs within the loop to exit
- A goto-statement, if the target of the goto-statement is outside of *b*

12.1.3.2 Implicit Unwinding

A statement-sequence *s* within local-block *b* terminates with an *implicit unwind* of *b* if there is an *implicit branch*¹ or other transfer of control (that terminates *s*) from within *s* to a point outside of *b* caused by procedural condition handling.

12.1.4 Interpretation of a Local Block

The execution of a local-block *b* proceeds as follows:

1. If *b* is a procedure body, then:
 1. Local parameters are bound to their corresponding arguments.
 2. If the procedure has a frame, the frame is established with a minimum chance of causing an exception.

¹ An implicit branch is a transfer of control similar to a jump caused by a goto-statement, but has no constraints on its target. The concept of an implicit branch is introduced only to define some of the more mysterious properties of procedural condition handling.

3. If the procedure specifies argument probing, the arguments to the procedure are validated. This will cause an exception if the arguments are not valid.
2. *B*'s declarations are interpreted to the extent necessary at run time. (Most of the work is done at compile time.) This can involve expression evaluation, the initialization of variables, and additional allocation of stack storage, so an exception can occur.
3. If *b* has an enable-section, the enable-section is interpreted. This can accomplish the following (in order):
 1. Enable an environment
 2. Disable alerts
 3. Lock a semaphore
 4. Establish a procedural condition handler
 5. Alter the state of underflow detection
4. *B*'s main statement-sequence is executed, producing a termination state *sterm* relative to *b*. The execution of the statement-sequence can include the invocation of *b*'s procedural condition handler (see Section 12.4).
5. If *b* has a procedural condition handler, the handler is annulled. Underflow detection reverts to the state in force before the execution of *b*.
6. If *sterm* is an exception for which *b* has a handler, the handler's main statement-sequence is executed and *hterm* is *continue*. Otherwise, *hterm* is *sterm*.
7. If *hterm* is *unwind* or an exception, and *b* has an unwind handler, the unwind handler is executed to produce a termination state *uterm* relative to *b*. It is an error with unpredictable consequences if *uterm* is not *continue*.
8. If *b* locked a semaphore, the semaphore is released. Alerts return to the state in force before the execution of *b*.
9. *B* terminates in one of the following ways:
 - If the main statement-sequence of *b* or the statement-sequence of one of *b*'s exception handlers terminates sequentially, then *b* terminates sequentially. If *b* is a procedure body, sequential termination of *b* causes an implicit return from the procedure; otherwise, sequential termination of *b* transfers control to the statement lexically succeeding *b*.
 - If *hterm* is an exception, *b* terminates by propagating *hterm* to *b*'s nearest *dynamic ancestor*.
 - If *b* has a procedural condition handler *pch*, and *pch* is invoked and then terminates with an exception *e*, *b* terminates by propagating *e* to *b*'s nearest dynamic ancestor.
 - Otherwise, *b* terminates with an explicit transfer of control from within either the main statement-sequence of *b* or one of *b*'s handlers to a point outside *b*. \ \ \ This can unwind enclosing local-blocks. \ \ \

Steps 1 through 3 constitute *b*'s *prologue*.

12.1.4.1 The Enable Section of a Local Block

■ enable-section

```
[ lock-enable ]  
[ alerts-enable ]  
[ underflow-enable ]  
[ condition-handler-enable ]  
[ message-vector-enable ]  
[ environment-enable ]
```

The order of the elements of an enable-section is fixed.

A local-block cannot have both a lock-enable and an alerts-enable.

12.1.4.1.1 Acquiring a Lock

■ lock-enable

```
LOCK data-reference ;
```

If an enable-section of a local-block *b* contains a lock-enable, then, during *b*'s prologue:

1. The data-reference is interpreted to obtain a value *sem* of type USER_SEMAPHORE.
2. Alerts are disabled for *b*.
3. *Sem* is locked.

When *b* terminates, *sem* is unlocked.

12.1.4.1.2 Disabling Alerts

■ alerts-enable

```
DISABLE ALERTS ;
```

If an enable-section of a local-block *b* contains an alerts-enable, then alerts are disabled during *b*'s prologue.

If *b* does not contain an alerts-enable, then the state of alerts during the execution of *b* is the same as for *b*'s dynamic ancestor.

When *b* terminates, alerts revert to the state in force prior to the execution of *b*.

12.1.4.1.3 Enabling and Disabling Underflow

■ underflow-enable

```
{ ENABLE }  
{ DISABLE } UNDERFLOW ;
```

If the enable-section of a local-block *b* contains an underflow-enable, then the detection of floating-point underflow is disabled or enabled (as indicated) during *b*'s prologue.

If *b* does not contain an underflow-enable, then the state of underflow detection during the execution of *b* is the same as for *b*'s dynamic ancestor.

When *b*'s main statement-sequence terminates, underflow detection reverts to the state in force before the execution of *b*.

Underflow detection is initially disabled when a program begins execution.

12.1.4.1.4 Enabling a Condition Handler

■ condition-handler-enable

ENABLE CONDITION HANDLER name ;

If the enable-section of a local-block *b* contains a condition-handler-enable, then the name must denote a procedure *p* with type `CONDITION_HANDLER`. The name is interpreted as a data-reference to obtain a procedure value *pval*. *Pval* is established as a procedural condition handler for *b* during *b*'s prologue.

When *b*'s main statement-sequence terminates, *p* is annulled as a procedural condition handler for *b*.

12.1.4.1.5 Enabling a Message Vector

■ message-vector-enable

ENABLE MESSAGE VECTOR name ;

If the enable-section of a local-block *b* contains a message-vector-enable, then:

- The name must denote an assignable data item *mv* of type `MESSAGE_VECTOR`.
- No local-block contained within *b* can have a message-vector-enable.
- Either *b* or a local-block contained within *b* must have an exception-handler-section.

Before the execution of an exception handler of or within *b*, information about the exception that caused the handler to be executed will be stored at *mv*'s location.

12.1.4.1.6 Enabling an Environment

■ environment-enable

ENABLE ENVIRONMENT name ;

If the enable-section of a local-block *b* contains an environment-enable, then the name must denote an environment *env*. Local-block *b* enables *env* (see Section 8.3.3).

12.2 Contents of the \$CONDITION Built-in Module

The \$CONDITION module is built into Pillar and contains the following declarations:

- The type `SEVERITY`
- The type `STATUS`, and associated built-in functions
- The type `MESSAGE_VECTOR`, and associated built-in functions
- The type `CONDITION_HANDLER`
- The type `CONDITION_DISPOSITION`
- The type `USER_SEMAPHORE`, and associated built-in functions

12.2.1 SEVERITY

SEVERITY is an enumerated type whose definition follows:

```
TYPE
    severity: (
        warning,
        success,
        error,
        informational,
        fatal,
        no_severity
    );
```

12.2.2 STATUS

A value of type STATUS contains a condition *c* and a value *sev* of type SEVERITY. *Sev* need not be equal to *c*'s SEVERITY value.

STATUS is a quadword-sized opaque type that receives special interpretation in the following contexts:

- STATUS cannot be revealed.
- The relational operators == and <> are defined on values of type STATUS. The STATUS values' conditions are compared for equality or inequality, while the STATUS values' SEVERITY values are ignored.
- A procedure with result type STATUS can be invoked as a procedure-call-statement (see Section 13.3.1).
- The built-in functions GET_SEVERITY and SUCCESS_STATUS operate on values of type STATUS.

12.2.2.1 GET_SEVERITY Built-In Function

```
result = GET_SEVERITY (status-value) ;
```

Argument

status-value. This argument is interpreted to obtain a value of type STATUS.

Result

This function returns the SEVERITY value of *status-value*.

12.2.2.2 SUCCESS_STATUS Built-In Function

```
result = SUCCESS_STATUS (status-value) ;
```

Argument

status-value. This argument is interpreted to obtain a value of type STATUS.

Result

This function returns a BOOLEAN value: TRUE if the severity of *status-value* is *severity.success* or *severity.informational*; FALSE otherwise.

12.2.3 MESSAGE_VECTOR

A value *mv* of type MESSAGE_VECTOR contains the following components:

- A message *m*
- A value of type SEVERITY
- An exception flag: TRUE if *mv* represents an exception, FALSE if *mv* represents a report, and undefined if *mv* represents an expanded message
- Arguments to *m*, which are string values to be substituted for *m*'s substitution parameters

MESSAGE_VECTOR is an opaque type that receives special interpretation in the following contexts:

- MESSAGE_VECTOR cannot be revealed
- Assignment to a data item of type MESSAGE_VECTOR is disallowed
- The result type of a procedure type cannot be MESSAGE_VECTOR
- The built-in functions GET_SEVERITY, GET_STATUS, GET_EXCEPTION_FLAG, SET_EXCEPTION_FLAG, SET_MESSAGE, EXPAND_MESSAGE_VECTOR, and COPY_MESSAGE_VECTOR operate on locations and values of type MESSAGE_VECTOR

12.2.3.1 GET_SEVERITY Built-In Function

```
result = GET_SEVERITY (vector) ;
```

Argument

vector. This argument is interpreted to obtain a value of type MESSAGE_VECTOR.

Result

This function returns a result of type SEVERITY, which is the severity code contained in *vector*.

12.2.3.2 GET_STATUS Built-In Function

```
result = GET_STATUS (vector) ;
```

Argument

vector. This argument is interpreted to obtain a value of type MESSAGE_VECTOR.

Result

This function returns a result of type STATUS, which contains *vector*'s condition and severity. A range violation occurs if *vector*'s severity is *severity.no_severity*.

12.2.3.3 GET_EXCEPTION_FLAG Built-In Function

```
result = GET_EXCEPTION_FLAG (vector) ;
```

Argument

vector. This argument is interpreted to obtain a value of type MESSAGE_VECTOR.

Result

This function returns a BOOLEAN result that is the value of *vector*'s exception flag.

12.2.3.4 SET_EXCEPTION_FLAG Built-in Function

SET_EXCEPTION_FLAG (*vector*, *boolean-value*) ;

Arguments

vector. This argument is interpreted to obtain an assignable location of type MESSAGE_VECTOR.

boolean-value. This argument is interpreted to obtain a BOOLEAN value.

Result

This function sets *vector*'s exception flag to the value given by *boolean-value*.

12.2.3.5 SET_MESSAGE Built-in Function

SET_MESSAGE (*vector*, *name*, [, *string-arguments...*]) ;

Arguments

vector. This argument is interpreted to obtain an assignable location of type MESSAGE_VECTOR.

name. This argument is the name of a message, or is interpreted to obtain a value of type STATUS.

string-arguments. This is a list of zero or more arguments to be interpreted to obtain a list of string values for use as substitution parameters for the message component of *name*. The number of string arguments supplied must equal the number of substitution parameters required by the message component of *name*.

This function sets the message-, severity-, and arguments-components of *vector*. If *name* denotes a message, the severity is *severity.no_severity*.

12.2.3.6 EXPAND_MESSAGE_VECTOR Built-in Function

EXPAND_MESSAGE_VECTOR (*vector*, *target*) ;

Arguments

vector. This argument is interpreted to obtain a value of type MESSAGE_VECTOR.

target. This argument is interpreted to obtain an assignable location of a string type.

Result

This function substitutes *vector*'s arguments into *vector*'s message, creating a string value *s*, which is assigned to *target*. A range violation occurs if the length of *target* is less than the length of *s*.

12.2.3.7 COPY_MESSAGE_VECTOR Built-In Function

`COPY_MESSAGE_VECTOR (source-vector, target-vector) ;`

Arguments

source-vector. This argument is interpreted to obtain a value of type `MESSAGE_VECTOR`.

target-vector. This argument is interpreted to obtain an assignable location of type `MESSAGE_VECTOR`.

Result

This function copies all components of *source-vector* to *target-vector*.

12.2.4 CONDITION_HANDLER

`CONDITION_HANDLER` is a procedure type defined as follows:

```
TYPE
  condition_handler: PROCEDURE (
    IN v: message_vector
  ) RETURNS condition_disposition;
```

12.2.5 CONDITION_DISPOSITION

`CONDITION_DISPOSITION` is an enumerated type defined as follows:

```
TYPE
  condition_disposition: (
    resignal,
    continue,
    resignal_exception
  );
```

12.2.6 USER_SEMAPHORE

A value of type `USER_SEMAPHORE` is a handle to a user-mode semaphore object.

`USER_SEMAPHORE` is an opaque type that receives special interpretation in the following contexts:

- `USER_SEMAPHORE` cannot be revealed.
- The built-in functions ??? operate on locations and values of type `USER_SEMAPHORE`.
\\\ *What operations are needed?* \\\

12.3 Messages, Conditions, Exceptions, and Reports

A *condition* is an *exception* or a *report*. A condition is an exception if it has a nonsuccess severity or is specially flagged as an exception; otherwise, it is a report.

A report that occurs during the execution of code within a (procedural condition, exception, or unwind) handler and outside of any local-blocks within the handler is promoted to an exception.

12.3.1 Promotion of a Condition

Condition *c* is promoted to exception *e* by the following rule:

If *c* is a report, *e* has all properties of *c* except that *e* is specially flagged as an exception; otherwise, *e* is *c*.

12.3.2 Message Declarations

■ complete-message-declaration

`[MESSAGE] identifier = character-string-literal ;`

The keyword `MESSAGE` must be present unless this complete-message-declaration immediately follows another complete- or external- message-declaration.

The identifier is declared as a message whose text is the character-string-literal. The character-string-literal can contain any number of exclamation point (!) characters, which represent substitution parameters of the message text.

■ external-message-declaration

`[MESSAGE] { identifier } ... EXTERNAL ;`

The keyword `MESSAGE` must be present unless this external-message-declaration immediately follows another complete- or external- message-declaration.

Each identifier is declared as a message, with a text to be supplied at link time.

12.3.3 Condition Declarations

A condition contains a message and a value of type `SEVERITY`. A condition declared by an external-condition-declaration also contains system-dependent information.

■ complete-condition-declaration

`[CONDITION] identifier : condition-severity [= character-string-literal] ;`

■ condition-severity

`{
 SUCCESS
 INFORMATIONAL
 WARNING
 ERROR
 FATAL
}`

The keyword `CONDITION` must be present unless this complete-condition-declaration immediately follows another complete- or external- condition-declaration.

The identifier is declared as a condition, with the severity indicated by the condition-severity, and the message text given by the character-string-literal. The character-string-literal can contain any number of exclamation point (!) characters, which represent substitution parameters of the message text. If the character-string-literal is omitted, the condition acquires a default message which has no substitution parameters.

■ external-condition-declaration

[CONDITION] { identifier } ,... EXTERNAL ;

The keyword **CONDITION** must be present unless this external-condition-declaration immediately follows another complete- or external- condition-declaration.

Each identifier is declared as a condition, with a severity and message text to be supplied at link time.

12.4 Procedural Condition Handling

A *condition* is an event that interrupts the normal control flow of a program. A condition is either a *report*, in which case a *procedural condition handler* can cause execution to resume at the point where the condition occurred, or an *exception*, which forces termination of the execution of the statement-sequence in which the condition occurred.

If a local-block *b* does not have a procedural condition handler, a report that occurs during the execution of code within the main statement-sequence *s* of *b* and outside of any local-blocks within *b* is promoted to an exception *e*, *s* terminates with an implicit unwind of *b*, and *e* is propagated to the nearest dynamic ancestor of *b*. If a condition is propagated to *b* from a dynamic descendant, the effect is as if *b* had a procedural condition handler whose only action was to return *resignal_exception*.

A procedural condition handler *pch* enabled for a local-block *b* with main statement-sequence *s* is invoked if a condition occurs while executing code that is within *s* and outside of any local-blocks within *b*, or if an immediate *dynamic descendant* of *b* propagates a condition to *b*. If a condition *c* occurring during the execution of statement-sequence *s1* results in the invocation of *pch*, the terminations of *s* and *s1* are defined in terms of the termination of *pch*.

- If *pch* terminates with an exception *e*, *s1* terminates with an implicit branch to the point within *pch* where *e* occurred.
- If *pch* terminates with a goto-statement with target label *lab*, *s1* terminates with an implicit branch to *lab*.
- If *pch* terminates by returning a **CONDITION_DISPOSITION** value *disposition*, then:
 - If *disposition* equals *resignal*, *c* propagates to *b*'s nearest dynamic ancestor, without terminating *s1*.
 - If *disposition* equals *continue*, and condition *c* is a report, the execution of *s1* resumes immediately after the point at which the condition occurred. It is an error with unpredictable consequences if *c* is an exception.
 - If *disposition* equals *resignal_exception*, condition *c* is promoted to exception *e*. If *s1* is not *s*, *s1* terminates with an implicit branch to the last point in *s*, which terminates with *e*.

12.5 Exception Handlers

■ exception-handler-section

WHEN exception-list THEN statement-sequence

■ exception-list

```
{ exception-name-list }  
{ exception-others }
```

For the local-block immediately containing the exception-handler-section, the statement-sequence is an exception handler for all conditions designated in the exception-list.

■ exception-name-list

```
EXCEPTION { name } ,...
```

Each name must be declared as a condition. A condition can be named only once in all exception-handler-sections of a local-block. An exception-name-list designates each condition named within it.

■ exception-others

```
EXCEPTION OTHERS
```

An exception-others designates all conditions not designated by other exception-lists in a local-block's exception-handler-sections. An exception-handler-section that contains an exception-others must be the last exception-handler-section in a local-block.

12.6 Unwind Handlers

■ unwind-handler-section

```
WHEN UNWINDING THEN statement-sequence
```

The statement-sequence is an unwind handler for the local-block immediately containing the unwind-handler-section. The statement-sequence must not contain either a goto-statement with a target outside the statement-sequence or a return-statement.

*\\\On some systems, most notably VMS, unwind handlers are not invoked if a program terminates with an exception. On such systems, it is advisable for programs that have unwind handlers to have an OTHERS exception handler for the main procedure. *

CHAPTER 13

PROCEDURES

This chapter describes Pillar procedures. Because the complete language for procedures is rather elaborate, the form of straightforward declarations and calls might not be immediately apparent from the syntax. A typical procedure returning a result and containing subprocedures has the following form:

```
PROCEDURE name (parameter declarations)
  RETURNS result description ;
  local declarations
BEGIN
  code
SUBPROCEDURES
  declarations of subprocedures
END PROCEDURE name ;
```

The forms of *parameter declarations* are explained in Section 13.2. Note that parentheses are present even if there are no parameters. The *result description* is explained in Section 13.3. If the procedure does not return a result, *RETURNS* and *result description* are omitted.

Together, a procedure's parameter declarations, its result description, its call linkage details, and its environment determine a procedure's type. In Pillar's syntax, all these occur within the category procedure-type-constructor.

A procedure heading always ends with a semicolon. The procedure's name is repeated at the end of the entire procedure declaration, which always ends with a semicolon.

The body is a local-block (see Section 12.1); it contains local declarations, code, and subprocedure declarations. In the local-block of a procedure, subprocedures cannot be declared in the local declarations position; they must be declared only after the SUBPROCEDURES keyword. Subprocedure declarations are described in Section 12.1.2.

A procedure is invoked by the interpretation of a procedure-invocation whose form is:

data-reference (arguments)

This can occur as a procedure-call-statement or in an expression as a procedure-function-reference, but the usage must be consistent with the procedure's result type (if any). Note that a procedure declared as returning STATUS can be invoked as a procedure-call-statement, as discussed in Section 13.3.1.

The parentheses are always required in a procedure-invocation, even when there are no arguments.

Interpreting the arguments of a procedure-invocation is performed in two steps. First the list of arguments is matched with the procedure's parameter list to determine, for each argument, the particular parameter to which it is passed. Pillar has rather flexible conventions that allow keyword notation, optional arguments (with or without default values), and list arguments (zero or more arguments passed to a single parameter). All this is described in Section 13.4.

Once the arguments are matched with parameters, each argument is interpreted in accordance with the parameter's mode (IN, OUT, IN OUT, or BIND) and data type. This yields a value in the case of an IN parameter, or an assignable location in the case of other parameter modes. The parameter is then bound to the value or location.

Once the arguments have been interpreted, the procedure's body (a local block) is interpreted. This is called a *procedure activation*. Within the procedure activation, a reference to a parameter denotes the value or variable to which the parameter was bound. (There are some other possibilities for parameter binding, as explained in Sections 13.2 and 13.4.1.)

13.1 Procedure Declarations

■ complete-procedure-declaration

```
[ PROCEDURE ] identifier procedure-type-specification  
  [ [ INLINE [ ONLY ]  
    WITH ARGUMENT PROBING ] ] ;  
local-block PROCEDURE identifier ;
```

The keyword **PROCEDURE** must be present unless the complete-procedure-declaration immediately follows another complete-procedure-declaration or external-procedure-declaration (see Section 13.6).

The identifier in a complete-procedure-declaration is declared as the name of a new procedure. The scope of the declaration is the entire module if the declaration is at module-level. The scope is the containing local-block if the declaration is in a subprocedure-section (described in Section 12.1.2). The procedure-type-specification describes the procedure's parameters and the type of its result (if it returns a result). The type specification can also control some details of the linkage used to implement calls to the procedure. All these topics and the use of **INLINE** are described in this chapter.

The local-block in the complete-procedure-declaration is the procedure's body. Note that the identifier giving the procedure's name is repeated following the local-block's terminating **END**, and the entire declaration ends with a semicolon.

13.1.1 Procedure Type Specifications and Constructors

■ procedure-type-specification

```
{ procedure-type-constructor }  
{ procedure-type-reference }
```

■ procedure-type-constructor

parameter-list [procedure-result-specification]
[ENVIRONMENT (name)]
[procedure-linkage-specification]

A procedure-type-constructor introduces a new procedure type. The properties of the type are given by the constructor's components (see Sections 13.2, 13.3, and 13.8).¹

■ procedure-type-reference

parameter-repetition OF TYPE name

The name must be the name of a procedure type, and the type reference denotes that type.

13.1.1.1 Parameter Repetitions

When a procedure type is used in a procedure declaration, or the body of a procedure is given in a procedure completion, the parameter-list is generally not visible to a reader of the procedure's body. For this reason, Pillar requires that the parameter's names be repeated where they can be seen by the reader.

■ parameter-repetition

([identifier] , ...) [RETURNS [identifier]]

A parameter-repetition occurs in a procedure-type-reference or a procedure-completion. In either case, an existing procedure type is being used. The identifiers in the parameter-repetition must be exactly the same (in number and order) as the names of the parameters of the procedure type. Note that parentheses are present even if there are no parameters.

If the procedure type being used returns a result, the keyword RETURNS must be present, as shown. In addition, if the procedure type has a *named* result, the identifier giving its name must be repeated following RETURNS.

Example of a parameter-repetition:

```
(addend1, addend2, addend3) RETURNS sum;
```

This parameter-repetition would occur, for example, in a procedure-completion for a procedure with this parameter-list and result-specification:

```
(IN addend1, addend2, addend3: integer) RETURNS sum: integer;
```

13.2 Parameters

This section contains the general syntax for parameter-lists. Subsections give the detailed rules for interpreting an argument in accordance with the declaration of its matching parameter. The rules for argument-parameter matching are in Section 13.4.1.

■ parameter-list

([parameter-declaration] ...)

¹ Possible future extensions to the language are preconditions and postconditions on procedures.

■ parameter-declaration

$$\left[\begin{array}{l} \text{IN} \\ \text{OUT} \\ \text{IN OUT} \\ \text{BIND} \end{array} \right] \{ \text{identifier} \}, \dots : \left\{ \begin{array}{l} \text{named-type} \\ \text{subrange-type-constructor} \\ \text{set-type-constructor} \end{array} \right\} \text{parameter-options};$$

The keywords IN, OUT, IN OUT or BIND (which specify parameter *modes*) can be omitted only if this parameter-declaration immediately follows the declaration of another parameter with the same mode. In other words, the occurrence of one of these keywords in a parameter-list determines the mode of all succeeding parameters that appear before the next occurrence of IN, OUT, IN OUT or BIND.

According to the syntax, each parameter-declaration terminates with a semicolon. However, the semicolon can be omitted in the last parameter-declaration in a parameter list (that is, in the parameter-declaration immediately preceding the right parenthesis terminating the list).

As an example of a single parameter-declaration, consider the declaration of an integer input parameter that must be supplied using keyword notation and that has a default value of 10:

```
IN count: integer = 10 KEYWORD;
```

A parameter-list occurs in a procedure-type-constructor that introduces a new procedure type. The identifiers in each of its parameter-declarations are declared as names of parameters of the procedure type. The scope of these declarations is the procedure-type-constructor together with all the procedure-type-specifications and procedure-bodies that depend on the type (these can be in other modules). The procedure type's parameter list contains all the parameters in the order in which their names occur in the original parameter-list.

Within a single parameter-declaration, all the parameters have the same type, which is given by the named-type, subrange-type-constructor, or set-type-constructor. Expressions within these type specifications must be simple expressions, which can refer to the values of IN parameters, the extents of a parameter of any mode, and certain other values that depend on the procedure's parameters. The exact rules are part of the definition of the class of simple expressions.

A parameter's type can be specified as a named-type containing matching extents. If so, the values of extents are derived from the argument matching the *first* parameter named in the list of identifiers; for example:

```
PROCEDURE p (  
    IN a, b: vector(*);  
    OUT c: vector(a.n));
```

The argument passed to *a* determines the value of the matching extent. Therefore, only *a* has matching extents; *b* does not. In this example, the extent's name is *n*, and all three parameters have the same type.

In contrast, consider the following example:

```
PROCEDURE p (  
    IN a: vector(*);  
    IN b: vector(*);  
    OUT c: vector(a.n));
```

In *this* example, the parameters *a* and *b* do *not* have the same type; rather, there are extents (*a.n* and *b.n*) derived from both *a* and *b*.

13.2.1 IN Parameters

An argument passed to an IN parameter is interpreted as an expression. Unless the IN parameter's type contains matching extents, the parameter's type is the target type for evaluating the argument expression. Thus, the rule for the argument's type is that it must be assignment compatible to the parameter's type, and conversion to the parameter's type can occur.

An IN parameter with matching extents does not provide a target type. The argument's type must match the parameter type as described in Section 13.2.5. The net effect is that the parameter will have the same type as the argument.

The type rules in the preceding paragraphs can be relaxed by use of the CONFORM parameter option.

An IN parameter is bound to the value produced by evaluating the argument. It is not possible to modify an IN parameter (attempting to modify an IN parameter through a dereferenced pointer is an error with unpredictable consequences). To avoid the cost of copying large values unnecessarily, a rule is imposed concerning the use of a variable as an argument passed to an IN parameter:

If an argument passed to an IN parameter is a data-reference to an assignable data item, and if the data-reference's data type is not a small type (see Section 5.16.5), then any modification of the data item's value during the procedure's execution is an error with unpredictable consequences.

As an example:

```
VARIABLE s : STRING(100);  
PROCEDURE p (IN x : STRING(*));  
  BEGIN  
    .  
    .  
    s = '';  
    .  
    .  
  END PROCEDURE p;  
  
p(s);
```

The behavior is unpredictable because *p* modifies the input argument *s*, which makes references to the parameter *x* unpredictable.

This error is uncommon, and can sometimes be diagnosed by the compiler. This error does not occur with parameters with small types, since they are always copied.

13.2.1.1 Special Restrictions on IN Parameters

The type of an IN parameter is not allowed to be any of the following types (which would make little or no sense in this context):

- VARYING_STRING
- BIT or BIT_DATA (this applies to all parameters)

13.2.2 OUT Parameters

An argument passed to an OUT parameter is interpreted as an assignable data-reference. The parameter's type must be assignment compatible to the argument's type.

An OUT parameter is bound to an assignable data item. If the procedure activation terminates normally (that is, with a RETURN), the parameter's current value is assigned to the argument. In other cases, the argument might not be modified. During the procedure's execution, accessing the argument (other than through the parameter) is an error with unpredictable consequences.

It is likely that the compiler will use a temporary variable if the parameter's data type is small, but use the argument directly in other cases. However, this depends on system conventions, data-type details, and the effects of such things as remote procedure calls.

13.2.3 IN OUT Parameters

An argument passed to an IN OUT parameter is interpreted as an assignable data-reference. Its type must be compatible with the parameter's type.

An IN OUT parameter is bound to an assignable data item, and the argument's current value is assigned to the variable. This assignment takes place before execution of the statement-sequence in the procedure's body. If the procedure activation terminates normally (that is, with a RETURN), the parameter's current value is assigned to the argument. In other cases, the argument might not be modified. During the procedure's execution, accessing the argument (other than through the parameter) is an error with unpredictable consequences.

As is the case for OUT parameters, it is likely that the compiler will use a temporary variable if the parameter's data type is small, but use the argument directly in other cases. However, this depends on system conventions, data-type details, and the effects of such things as remote procedure calls.

13.2.3.1 STRING and VARYING_STRING, OUT and IN OUT Parameters

For STRING and VARYING_STRING OUT and IN OUT parameters, the corresponding arguments must meet the following requirements, in addition to being compatible with the parameter:

- If the parameter's type is STRING(n), the argument's type must also be STRING(n).
- If the parameter's type is VARYING_STRING(n), the argument's type must also be VARYING_STRING(n).
- If the parameter's type is STRING(*), the argument's type must be STRING(n).
- If the parameter's type is VARYING_STRING(*), the argument's type must be VARYING_STRING(n).

The purpose of these rules is to allow the compiler to avoid making copies that are unnecessary in the usual usage of these parameters.

13.2.4 BIND Parameters

An argument passed to a BIND parameter is interpreted as an assignable data-reference. Its type must be equivalent to the parameter's type. The parameter is bound to the assignable data-item denoted by the argument. The argument can be accessed by other means during execution of the procedure, and it can be shared with other threads of execution. A BIND parameter should only be used when some sort of shared or indirect access is desired.

In a parameter declaration of a BIND parameter, the parameter's type can be given only as a named-type. A subrange-type-constructor or a set-type-constructor is not allowed.

13.2.5 Matching Extents

A parameter's type can contain matching extents, whose values will be determined by corresponding extents in the argument's type; for example:

```
PROCEDURE reverse (IN s : STRING(*)) RETURNS STRING(s.LENGTH);
```

Matching extents can be used in other parameter types, and a parameter can have some matching extents and some other extents; for example:

```
TYPE vector (n : INTEGER[1..]) : ARRAY[1..n] OF REAL;
   matrix (m, n : INTEGER[1..]) : ARRAY[1..m, 1..n] OF REAL;

VARIABLE row : vector(10); col : vector(20);
   t : matrix(10, 20);

PROCEDURE p (IN x : vector(*);
             IN a : MATRIX(x.n, *);
             OUT y : vector(a.n));

BEGIN
  .
  .
  .
END PROCEDURE p;

p (row, t, col);
```

The procedure *p* multiplies a vector by a matrix to obtain another vector. The matrix's row dimension must equal the number of elements in the input vector, and its column dimension must equal the number of elements in the output vector. Thus, only two extents need to be determined by matching. In the illustrated invocation of *p*, *x.n* and *a.m* are equal to 10; *a.n* and *y.n* are equal to 20.

The example has the matrix parameter *a* with one matching extent and one nonmatching extent, just to illustrate this possibility. It is more natural to declare *y* as *vector(*)* and *a* as *matrix(x.n, y.n)*.

Note that there must be exactly one asterisk (in the parameter declaration) for each matching extent.

13.2.5.1 The Normal Extent-Matching Rule

Let x be a non-CONFORM parameter whose type is $t(e_1, \dots, e_n)$, where some or all of the e_i are matching extents (asterisks). If an argument y is passed to x , then y must also be of type t , so y 's type has the form $t(v_1, \dots, v_n)$. A matching extent e_i in x 's type takes on the value v_i from y 's type. Any nonmatching extent e_j must be equal to the corresponding extent v_j (otherwise, range violation). The net result is both x and y have type $t(v_1, \dots, v_n)$.

The type of the argument y does not have to be exactly t ; it can be another flexible type derived from t ; for example:

```
TYPE
  square_matrix (m : integer[1..] : matrix(m, m));
VARIABLE
  y: square_matrix(10);
PROCEDURE p (IN x: matrix(*, *));
BEGIN
  .
  .
  .
END PROCEDURE p;

p(y);
```

The exact type of y is *square_matrix(10)*, but this is derived from *matrix(10,10)*, so y is compatible with the parameter x .

Matching also applies to POINTER types. Thus, if x is a non-CONFORM parameter whose type is POINTER $t(e_1, \dots, e_n)$, an argument, y , passed to x must have type POINTER $t(v_1, \dots, v_n)$, and each e_i matches v_i . As with non-POINTER types, y 's type can be derived from POINTER t ; for example:

```
TYPE
  node_type_code : (symbol_node_code, token_node_code, ...);
  node (n_type : node_type_code) :
    RECORD
      VARIANTS CASE n_type
        WHEN symbol_node_code THEN ...
        WHEN token_node_code THEN ...
        .
        .
      END VARIANTS;
    END RECORD;
PROCEDURE p (IN q : POINTER node(*));
TYPE
  symbol : POINTER node(symbol_node_code);
VARIABLE
  x : symbol;
BEGIN
  .
  .
  p(x); ! Within p, q.n_type is equal to symbol_node_code
  .
  .
END PROCEDURE p;
```

13.2.5.2 Extent Matching for CONFORM Arrays

The CONFORM option can be specified for an array parameter whose type has matching extents. In this case, if an argument does not satisfy the normal extent matching rule, the compiler will determine a matching extent by comparing the argument's array type with the parameter's type treated as a symbolic array type. This language feature permits expressing general array manipulation routines in Pillar; for example:

```
MODULE array_routines;
  TYPE real_array (n : INTEGER[0..]) : ARRAY [1..n] OF REAL;
  PROCEDURE real_array_sum (IN x : real_array(*) CONFORM)
    RETURNS sum : REAL;
  BEGIN
    sum = 0;
    FOR i = 1 TO x.n LOOP
      sum = sum + x[i];
    END LOOP i;
    RETURN;
  END PROCEDURE real_array_sum;
```

This routine sums any one-dimensional array whose element type is REAL. Programmers using this routine can ignore the type *real_array* in calls:

```
PROCEDURE main();
  TYPE
    my_array_type: ARRAY [1..17] OF real;
  VARIABLE
    my_array: my_array_type;
    x: real;
  BEGIN
    .
    .
    .
    x = real_array_sum(my_array);
    .
    .
  END PROCEDURE main;
```

Programmers can also ignore the detailed definition of the CONFORM array feature. The general idea is that the compiler expands the parameter's declared type (a bound-flexible-type) into its base array type yielding a type of the form:

```
ARRAY [lb1..hb1, ... lbn..hbn] OF elt-typ
```

Each matching extent, *e*, in the parameter type must occur in at least one of the following forms in the expanded type:

1. *lb_i..e*, where *lb_i* is constant
2. *lb_i..e*, where *lb_i* is not constant
3. *e..hb_i*

An argument passed to the parameter must be an array with the same number of dimensions and an equivalent element type. However, CONFORM also applies to the parameter's element type if it is exactly BYTE, WORD, LONGWORD, QUADWORD, or blank_DATA.

Let the argument's array type be:

ARRAY [$ly_1..hy_1, \dots, ly_n..hy_n$] OF *elt-typ*

If a matching extent, *e*, occurs in form 1 in the expanded parameter type, its value is determined by:

$$\text{ORD}(e) = \text{ORD}(hy_i) - \text{ORD}(ly_i) + \text{ORD}(lb_i)$$

The effect is just to make *e* consistent with the number of elements in the corresponding argument dimension. This is the most convenient value of *e*, because the declaration of the procedure determines the origin for indexing within the procedure.

If a matching extent *e*, occurs in form 2 or 3 in the expanded parameter type, its value is the corresponding argument bound, hy_i or ly_i , respectively.

If a matching extent occurs in a suitable form in more than one place in the expanded array type, it can be determined from any one of the occurrences. The resulting parameter array type must have the same number of elements as the argument type in each dimension (otherwise, range violation). If this is true, all computations of the extent yield the same value.

13.2.6 Parameters with Captured Extents

As explained in Section 5.11.1, a flexible record type, *t*, can have "captured extents." The values of these extents are actually stored in the record, so when *t* is used as a parameter type, the extents need not be specified.

If an extent of a flexible type is captured, it cannot be a matching extent in a parameter type. (Allowing such a matching extent would provide no additional capability.) Nor can the value of the extent be specified in the parameter-list.

13.2.7 Parameter Options

Parameter options are used to extend and modify the basic rules for argument-parameter matching and argument interpretation. The effects are described in a separate subsection for each option.

■ parameter-options

[CONFORM] [[parameter-default-value
OPTIONAL
LIST [[range-specification]]]] [KEYWORD] [STATUS VECTOR]

13.2.8 The CONFORM Option

The CONFORM option is used to relax the normal type compatibility rules in argument interpretation. It is useful for library routines and for routines that perform specific functions whose implementation depends on data representation.

One use of the CONFORM option is with array parameters; this use is described in Section 13.2.5.2. The other cases in which the CONFORM option can be used are as follows:

- The parameter's type is BYTE, WORD, LONGWORD, or QUADWORD.

The argument's type must have a constant size, and the same size in bits as the parameter (otherwise, a range violation occurs). If the parameter's mode is IN, OUT, or IN OUT, there is no alignment requirement on the argument. If the parameter's mode is BIND, the argument's known alignment must be at least as great as the parameter's.

- The parameter's type is `blank_DATA(*)`.

The argument's known alignment must equal or exceed the alignment requirement of the `blank_DATA` type. The argument's size in bits must be an exact multiple of 8, 16, 32, or 64, in accordance with the particular `blank_DATA` type (otherwise, a range violation occurs).

- The parameter's type is `blank_DATA(n)`.

The rule is the same as for `blank_DATA(*)`, except that the argument's size in bits must be exactly the same as the parameter's (otherwise, a range violation occurs).

- The parameter's type is `POINTER blank_DATA(*)`.

The argument's type must be `POINTER t` where t is not `ANYTYPE`. The size of t in bits must be an exact multiple of 8, 16, 32, or 64, in accordance with the particular `blank_DATA` type (otherwise, a range violation occurs). The alignment requirement of t is restricted according to the parameter's mode as follows:

- For an IN parameter, t 's alignment requirement must equal or exceed the alignment requirement of the `blank_DATA` type.
- For an OUT parameter, t 's alignment requirement must be less than or equal to the requirement of the `blank_DATA` type. This ensures that a pointer value assigned to the OUT argument will point to storage with at least the expected alignment.
- For a BIND or IN OUT parameter, t 's alignment requirement must be the same as the requirement of the `blank_DATA` type.

- The parameter's type is `POINTER blank_DATA(n)`.

The rules are the same as for `POINTER blank_DATA(*)` except that the size of the argument's type must be exactly the same in bits as the `blank_DATA` type.

- The parameter's type is `POINTER ANYTYPE`.

The argument's type can be any `POINTER` type.

13.2.9 Parameter Default Values

A default value can be specified only for an IN parameter; for example:

```
IN count : INTEGER[0..] = 0;
```

A parameter declared with a default value must have a constant type.

If an IN parameter has a default value, and a procedure-invocation does not contain an argument matching the parameter, the compiler uses the default value. Note that the parameter is not considered to be optional in the Pillar sense (and it does *not* appear as optional to the called procedure) because it has a value in every activation of the procedure.

■ parameter-default-value

= initializer

13.2.10 OPTIONAL Parameters

A parameter can be specified as `OPTIONAL`; for example:

```
OUT second_message : STRING(80) OPTIONAL;
```

A parameter specified as `OPTIONAL` cannot have matching extents.

If a procedure-invocation does not contain an argument matching an `OPTIONAL` parameter, the parameter is not bound to an argument during the procedure's invocation.

In general, within a procedure's code, it is a range violation to interpret a data-reference to an `OPTIONAL` parameter that is not bound to an argument. The only exception to this rule is when the optional parameter occurs as the argument to the `ARGUMENT_PRESENT` built-in function. (This function is used to test whether the parameter is bound to an argument.)

Note that if an `OPTIONAL` parameter occurs as an argument in a procedure-invocation, and is matched with an `OPTIONAL` parameter in the invoked procedure, the first `OPTIONAL` parameter *must* be bound to an argument (otherwise, a range violation occurs). This is true even though the second parameter is itself `OPTIONAL`.

An `OPTIONAL` parameter cannot be referred to within its containing procedure type or within the local declarations of a procedure body governed by that type. The purpose of this rule is to prevent the occurrence of exceptions (or unpredictable effects) in a block's prologue code.

13.2.11 LIST Parameters

A parameter can be specified as corresponding to a list of arguments by following its declaration with the keyword `LIST` followed by an optional range-specification; for example:

```
PROCEDURE p (IN items : POINTER t LIST [2..5]); ! A list of 2 to 5 pointer values
```

The range-specification, if present, must be a constant range-specification. It specifies the number of arguments, corresponding to the `LIST` parameter, that must be supplied in each call to the procedure (if an incorrect number of arguments is supplied, a range violation occurs). The range-specification is interpreted with a target type of `INTEGER [0..]`.

If the range-specification is not present, then the parameter corresponds to a list of zero or more arguments in any call to the procedure.

Within a procedure's code, three forms of references to a list parameter, *lst*, are allowed:

- The reference *lst.length* yields the number of arguments in the list of arguments to which the parameter is bound.
- The reference *lst[n]* is a reference to the *n*th argument in the list. It has the same properties as a reference to a non-`LIST` parameter whose mode and type are the same as those of the `LIST` parameter. This form of reference results in a range violation if *n* is less than one or greater than the number of arguments in the list.
- The reference *lst* is a reference to the entire list parameter. It can be used only to pass the entire parameter as an argument to another list parameter, *lst1*. Pillar allows this only if the following conditions hold:
 - *lst* and *lst1* have equivalent types.

- *lst* and *lst1* have the same mode.
- *lst* and *lst1* are passed by the same mechanism. That is, if either is declared with a linkage option, they must both be declared with that option.
- Neither *lst* nor *lst1* is specified with the TRAILING LIST linkage option (see Section 13.8.2).
- Passing the entire list parameter *lst* is equivalent to passing all of the arguments *lst[1]* through *lst[*lst.length*]*.

A LIST parameter cannot be referred to within its containing procedure-type-constructor or within the local declarations of a procedure body governed by that type (but *lst.length* is legal in these contexts). The purpose of this rule is to prevent the occurrence of exceptions (or unpredictable effects) in a block's prologue code.

A LIST parameter cannot have matching extents.²

13.2.11.1 Argument Interpretation for LIST Parameters

Section 13.4.1 explains how arguments are associated with a LIST parameter. Once the parameter is matched with a list of arguments, each argument in the list is interpreted in accordance with the LIST parameter's mode (IN, OUT, etc.) and type; for example:

```
PROCEDURE p (IN locations: integer LIST;  
            OUT items : POINTER t LIST);  
p (i1, i2, i3, items = q1, q2, q3);
```

Here the arguments *i1*, *i2*, *i3*, are interpreted as integers, while *q1*, *q2*, *q3* are interpreted as pointer variables.

13.2.12 KEYWORD Parameters

Normally, an argument in Pillar can be supplied using positional or keyword notation. (The syntax for both of these is shown in Section 13.4.1.) As an example, consider the procedure declaration:

```
PROCEDURE sample(IN index: integer;  
                IN count: integer) EXTERNAL;
```

One could supply the arguments to *sample* positionally:

```
sample(i1, c1);
```

or using keyword notation:

```
sample(index = i1,  
       count = c1);
```

or using a combination of the two notations, according to the rules in Section 13.4.1.

Pillar also provides the KEYWORD option, which allows the author of a procedure to restrict the coding style used in a call to the procedure. Declaring a parameter *x* with the KEYWORD option means that in calls to that procedure, arguments for *x* and all parameters following it in the parameter-list must be supplied, if at all, using keyword notation; positional notation

² A possible future extension to the language is to allow matching extents on LIST parameters. When this is done, the option MATCH INDIVIDUALLY can also be added, to signify that each argument in the list can have different extent values.

is never permitted for these arguments. Therefore, if the example above were declared this way:

```
PROCEDURE sample (IN index: integer KEYWORD;  
                 IN count: integer) EXTERNAL;
```

the call shown above using positional notation would not be valid.

The **KEYWORD** option itself can appear only once in a given parameter-list. Specifying it more than once would be a confusing redundancy, since it applies to the parameter it is specified with and all following parameters.

13.2.13 STATUS VECTOR Parameters

As discussed in Section 13.3.1, a procedure declared as returning **STATUS** can be invoked in a procedure-call-statement (thus not making use of the return value). If such a procedure should return a nonsuccess status, this status will be signaled as an exception. In this case, the **STATUS VECTOR** option can be used to obtain information on the condition.

The **STATUS VECTOR** option can be applied only to an **OUT** parameter of type `$condition.message_vector` of a procedure that is declared as returning **STATUS**. If a procedure is declared with a **STATUS VECTOR** parameter, and invoked in a procedure-call-statement that returns a nonsuccess status, the entire message vector contained in the **STATUS VECTOR** parameter is signaled.

Pillar enforces the following restrictions on **STATUS VECTOR** parameters:

- **STATUS VECTOR** can be applied only to *one* **OUT** parameter of type `$condition.message_vector` (described in Section 12.2.3).
- If one of a procedure's parameters has the **STATUS VECTOR** option, then the procedure must return a result and the result type must be **STATUS**.
- If one of a procedure's parameters has the **STATUS VECTOR** option, the procedure cannot have a result-variable. (The **STATUS VECTOR** parameter itself serves as the result variable.)
- If one of a procedure's parameters has the **STATUS VECTOR** option, return-statements within the procedure cannot specify a returned value. (Because the result value is taken from the **STATUS VECTOR** parameter.)
- Pillar makes the same checks about assignment to the **STATUS VECTOR** parameter as it would for the result-variable of a normal procedure. (These checks are described in Section 11.14.)

The parameter with the **STATUS VECTOR** option is optional in any call. Note that (unlike **OPTIONAL** parameters in Pillar) references to this parameter from inside the called procedure will not cause an access violation if no argument was supplied, since the compiler always provides a "dummy" argument in this case.

Here is an example of a usage of the **STATUS VECTOR** parameter option:

```
PROCEDURE foo(  
    OUT mv: MESSAGE_VECTOR STATUS VECTOR  
)  
    RETURNS STATUS;  
  
BEGIN  
    :  
    :  
    SET_MESSAGE(mv, ...);  
    :  
    :  
    RETURN;  
END PROCEDURE foo;
```

13.3 Procedure Results

The result type of a procedure is specified following the parameter list of a procedure-type-
constructor; for example:

```
PROCEDURE p (...) RETURNS SET integer [0..31];
```

If a procedure has a result type, it must return a value of that type; if it does not have a
result type, it cannot return a result. A name can be declared for the result; for example:

```
PROCEDURE p (...) RETURNS output : LONGWORD_DATA(n);
```

Within the procedure's code, *output* denotes a variable, and the value of *output* will be
used as the procedure's returned value unless explicitly overridden by an expression in the
return-statement.

■ procedure-result-specification

$$\text{RETURNS [identifier :] } \left\{ \begin{array}{l} \text{named-type} \\ \text{subrange-type-constructor} \\ \text{set-type-constructor} \end{array} \right\}$$

The named-type, subrange-type-definition, or set-type-definition gives the procedure result
type. The expressions within the procedure-result-specification must be simple expressions,
which can refer to the values of required IN parameters and the extents of a required
parameter of any mode. The exact rules are part of the definition of the class of simple
expressions.

If present, the identifier in a procedure-result-specification is declared as a "procedure result
name." The scope of the declaration is the procedure-type-constructor together with all the
procedure-type-specifications and procedure-bodies that depend on the type. (These can be
in other modules.)

The procedure result name cannot be referred to within the procedure-type-constructor or
local declarations of any procedure of the type having that named result.

13.3.1 Procedures Returning STATUS

If a procedure is declared as "RETURNS status" it gets special treatment in Pillar. Unlike
other result returning procedures, it can be invoked by a procedure-call-statement, for
example:

```
create_object(x);
```


If this is done, and the procedure called returns a nonsuccess status, an exception is signaled. The exception signaled is the condition denoted by the status value. This feature of Pillar allows one to construct routines (system services, for example) that return status values rather than signal exceptions. This feature can be used in conjunction with the STATUS VECTOR option on a parameter, which is described in 13.2.13.

A procedure returning STATUS can also be invoked as a procedure-function-reference.

13.4 Procedure Invocations

A procedure-invocation can occur as a procedure-function-reference or as a procedure-call-statement. It specifies the procedure to be invoked and the arguments (if any are supplied in the invocation) to be passed to it.

■ procedure-invocation

data-reference ([argument-list])

Note that the parentheses must be present even if the argument-list is empty.

To determine the procedure to be invoked, the data-reference is interpreted, and this also gives a type for the procedure. The type has a parameter list, and it has a result type if the procedure is one that returns a value.

Once the procedure's type is known, the arguments (if any) are interpreted in two steps. First, arguments are associated with parameters as described in Section 13.4.1. Second, each argument is interpreted in accordance with the parameter to which it is passed. This second step determines the values and variables to which the parameters are bound. If an argument is matched with an IN parameter, it is interpreted as an expression. For the other kinds of parameters (OUT, IN OUT, and BIND), an argument must be a data-reference and it is interpreted as an assignable reference. (The detailed rules for argument interpretation are given in Section 13.2.)

Once the arguments have been interpreted, the procedure's body (a local-block) is interpreted. During interpretation of the body, a reference to a local parameter of the procedure denotes the item bound to the parameter by the argument interpretation.

Execution of a return-statement or encountering the final END in the procedure terminates the procedure-invocation normally. If the procedure is one that returns a result, and the procedure-invocation occurs as a procedure-function-reference, the result value becomes the value of the procedure-function-reference.

The methods used by the compiler to implement a procedure-invocation depend on the circumstances. If the reference specifying the procedure is simply a name denoting an inline procedure or any procedure whose body is in the current module, the compiler can compile code specialized to the properties of the particular procedure. In other cases (EXTERNAL procedures not in the current module, and general procedure values), the compiler uses a standard call.

13.4.1 Argument Lists

This section explains how the parameters in a parameter list are matched with arguments in an argument list.

■ argument-list

$\left[\begin{array}{l} \text{argument} \\ \text{keyword-argument} \end{array} \right], \dots$

■ argument

expression

■ keyword-argument

identifier = { argument } , ...

In a keyword-argument, the identifier must be the name of one of the called procedure's parameters.

Parameters to which the **KEYWORD** option (see Section 13.2.12) applies must be supplied, if at all, as a keyword-argument. Other parameters can be supplied in this fashion, but are not required to be.

In an argument-list, any positional (nonkeyword) arguments must precede any keyword-arguments. The positional arguments are matched with parameters in the order in which the arguments and parameters occur in their respective lists. Unless a parameter has the **LIST** option, each parameter matches one argument until the list of positional arguments is exhausted.

If the positional matching reaches a **LIST** parameter, and there remains one or more unmatched positional arguments, all the remaining positional arguments are combined into one list that is matched with the **LIST** parameter. At this point, there must not be any unmatched positional arguments.

If the argument-list contains any keyword-arguments, they are now matched with parameters.³

The name in a keyword-argument must be the name of a parameter that has not already been matched with an argument. If only one argument follows the "name =", the parameter is matched with that; if there is more than one, the parameter must have the **LIST** option. All of the arguments within the keyword-argument are combined into one list that is matched with the **LIST** parameter.

At this point, all arguments have been matched, or an argument that does not match any parameter has been found (an error). If there are any unmatched parameters, they must have a default value, be **OPTIONAL**, or be **LIST** parameters. If a **LIST** parameter has been matched with no arguments or with a single argument, it is treated as being matched with a list of zero or one arguments, respectively, unless the single argument it was matched with is itself a **LIST** parameter.

³ The compiler does this in the order of their occurrence in the argument list, but the order does not really matter.

13.4.2 Argument List Validation

If a procedure is declared with the keywords **WITH ARGUMENT PROBING** preceding the declarations in the procedure's body, the procedure's argument list is validated when it is invoked. Pillar provides this primarily for executive entries in operating systems. Argument list validation is built into the language because the validation involves calling mechanisms that are hidden when one uses a higher-level language such as Pillar.

For example, the following procedure completed below will have its arguments validated on invocation:

```
PROCEDURE k$foo(n, y, z);  
WITH ARGUMENT PROBING;  
VARIABLE s : POINTER STRING(n);  
....  
BEGIN  
....  
END PROCEDURE k$foo;
```

The following restrictions on procedures declared with the **WITH ARGUMENT PROBING** option are enforced by Pillar:

- The procedure must not be declared **INLINE** or **INLINE ONLY**.
- The procedure must be one that returns **STATUS**.
- The **TRAILING LIST** linkage option (see Section 13.8.2) is not allowed for a parameter.
- If an **IN** parameter's data type is small (see Section 5.16.5), the **REFERENCE** linkage option (see Section 13.8.2) is not allowed. (This avoids having to capture small by-reference input values).

When **WITH ARGUMENT PROBING** is used, the procedure's argument list is probed, checked, and captured. If any violation is detected, the procedure immediately returns with an appropriate failure status code. Any random exception during this process will also cause immediate return with a failure status. Pillar will also set a "STATUS VECTOR" output parameter if the procedure has one. ⁴

The argument probing and capturing is done before execution of any prologue code related to the procedure's local declarations. The first two steps below capture the nonregister part of the argument list⁵ in the procedure's local storage. The remaining steps validate individual arguments.

1. If part of the procedure's argument list is in memory, that part is probed for read access. (All probes are made with respect to the previous mode as specified in the TCB.) The argument list itself is then copied into local storage (that is, into the procedure's stack frame). If the list contains any immediate **OUT** or **IN OUT** parameters, the memory occupied by the list will be probed for read/write access rather than read access. Note that this step handles all of the immediate arguments in the in-memory argument list. Any data referred to by those arguments will be probed in a later step (for example, arguments passed by reference).
2. If an argument is passed by descriptor, then the descriptor is validated for quadword alignment, probed for read access, and captured.

⁴ **WITH ARGUMENT PROBING** might not cause any code to be generated for some architectures. Special requirements on the run-time environment required by procedures specified using **WITH ARGUMENT PROBING** are to be specified.

⁵ Which part of the argument list is passed in registers is system dependent.

3. If an argument is a LIST argument, the list is probed for read access, validated for longword alignment and copied into local storage. Elements of the list (if any) are then processed (next three steps) as individual arguments.
4. If an IN argument is passed by reference, it is probed for read access, but it is not captured, and hence can still change. (The immediate IN values have all been captured by the preceding steps.) OPTIONAL IN parameters are only probed if they are present.
5. OUT and IN OUT parameters not passed immediately are probed for read/write access. OPTIONAL parameters in this category are only probed if they are present.
6. BIND parameters are probed for read/write access.
7. In all cases where an argument's memory is probed, the address is first checked for correct alignment according to the argument's data type. Note that the alignment of lists and descriptors is also checked during steps 1, 2, and 3.

As part of the procedure's common return code, any immediate OUT or IN OUT parameters that are not in registers are copied back into the original in-memory lists.⁶

13.5 INLINE and INLINE ONLY Procedures

As shown in Section 13.1, a procedure can be declared as `INLINE` or `INLINE ONLY`. Note that this is a property of a particular procedure, not of its procedure type. If the reference in a procedure-invocation denotes a procedure declared as `INLINE`, the compiler has the option of expanding the procedure's body into code at the point of the call. If the reference in a procedure-invocation denotes a procedure declared as `INLINE ONLY`, the compiler always performs this expansion at the point of the call.

Procedures declared with either `INLINE` or `INLINE ONLY` are referred to as *inline procedures*.

Here is an example of an inline procedure declaration:

```
PROCEDURE positive_result(  
    IN cli_status: integer)  
    RETURNS boolean  
    INLINE;  
  
BEGIN  
    RETURN  
        (local_flag AND (cli_status == cli$_locpres)) OR  
        (NOT local_flag AND ((cli_status == cli$_present) OR  
            (cli_status == cli$_defaulted)));  
END PROCEDURE positive_result;
```

The use of inline procedures will generally decrease program execution time because of the elimination of the general procedure-call overhead, and because the compiler's general optimization methods apply across the expanded code. Inline procedures are especially good for small procedures and functions, but it can be used advantageously on large ones as well. A good practice is to compare execution times of inline and noninline procedures, or simply inspect their resulting code.

A procedure declared as `INLINE` is subject to the following restrictions:

- Its body must not declare any subprocedures.

⁶ Pillar's argument list validation can be extended beyond this.

- Its body must not specify **WITH ARGUMENT PROBING**.

A procedure declared as **INLINE ONLY** is subject to all the restrictions imposed on procedures declared **INLINE**, plus the following restrictions:

- A reference to the procedure cannot be used for any purpose except to invoke the procedure.
- The body of the procedure cannot refer to the procedure, either directly or through a chain of invocations of other inline procedures. This would generate an infinite sequence of in-line code expansions. Note that an inline procedure, *p*, can invoke a noninline procedure that then invokes *p*.

An inline procedure's type can have a linkage specification, but this has no effect on the in-line code expansion.

An inline procedure does not appear in the call stack. Aside from this, and the restrictions checked by the compiler, **INLINE** and **INLINE ONLY** have no effect on the meaning of a procedure.

The use of inline procedures can cause difficulties in debugging because there is no stack frame for an inline procedure and no debugger symbol table information for the expanded procedure (variables or statement locations). The **NOINLINE** qualifier can be used on the **Pillar** command, to force inline procedures to be compiled as normal procedures, with stack frames and with debugger symbol table information. The **NOINLINE** qualifier applies only to inline procedures declared in the compilation unit, not to those defined in other modules.

13.6 EXTERNAL Procedures

At the module-level, a procedure can be declared as **EXTERNAL**. In this case, the module-level of some module (usually a different module) completes the definition of the procedure by use of a procedure-completion, which contains the procedure's actual body.

■ external-procedure-declaration

```
[ PROCEDURE ] identifier procedure-type-specification  
[ INLINE [ ONLY ] ] EXTERNAL ;
```

The keyword **PROCEDURE** must be present unless the external-procedure-declaration immediately follows another complete-procedure-declaration or external-procedure-declaration.

This declaration is allowed only in a procedure-section at module-level. The declaration's scope is the entire module.

13.6.1 Procedure Completions

■ procedure-completion

```
[ PROCEDURE ] identifier parameter-repetition ; [ WITH ARGUMENT PROBING ; ] local-block  
PROCEDURE identifier ;
```

WITH ARGUMENT PROBING is discussed in Section 13.4.2

A procedure-completion is allowed only in an implementation module at module-level. The identifier must be the name of an EXTERNAL procedure specified in an implement-section of the module. The local-block is the procedure's body. Note that the identifier giving the procedure's name must be repeated following the local-block.

For example, this procedure completion:

```
get_next_file(root,  
              context,  
              handle);  
  
BEGIN  
.  
.  
.  
END PROCEDURE get_next_file;
```

is a completion for this declaration:

```
PROCEDURE get_next_file(IN root: POINTER ANYTYPE CONFORM;  
                       IN context: integer;  
                       OUT handle: longword) EXTERNAL;
```

13.7 Environments and Procedures

If a procedure-type-constructor contains ENVIRONMENT(*name*), then *name* must be the name of an environment. This syntax causes all procedures that are instances of the type defined by the procedure-type-constructor to require the named environment. (Environments are discussed in Section 8.3).

13.8 Procedure Linkages

The methods used to pass arguments from a call site (that is, from the point of a procedure-invocation) to a called routine (that is, to the code of the invoked procedure) are inherently system dependent. Also, regardless of system conventions, the compiler can use different methods for code optimization if the compiler has control of the code for the called routine and for all call sites.

The conventions described in this section give only the general Pillar approach.

13.8.1 Argument Passing Mechanisms and Pillar Conventions

The term *argument item* means an item passed from the call site to the called routine, to represent the argument that is to be associated with a parameter in the called procedure. Typically, a system has some standard convention for passing an argument list, a list of argument items that correspond in order to parameters of the called procedure. In general, the nature of an argument item is determined by which of four basic argument-passing mechanisms is used for its parameter:

- Immediate mechanism—The parameter is directly associated with the argument item; for example, if the parameter is an INTEGER IN parameter, the argument item will be an integer value.
- Reference mechanism—The argument item is an address. The parameter is associated with the addressed data item or, in the case of an IN parameter, with the addressed item's value.

- **Descriptor mechanism**—The argument item is the address of a descriptor. The descriptor contains an address item (as in the reference mechanism) and information about the item's type, typically information derived from extent values for an argument passed to a parameter with a flexible type.
- **List mechanism**—There are two argument items: The address of a list of n element items and a count, n ($n \geq 0$). The elements themselves are otherwise like nonlist items in the main argument list.

To the extent consistent with the system, Pillar uses the following conventions, but they can be modified by a procedure-linkage-specification in a procedure type:

1. For a non-BIND parameter whose data type is small, the argument is passed by the immediate mechanism, if that is consistent with system conventions ⁷; otherwise, the reference mechanism is used.
2. If a parameter has a flexible type with n matching extents, these are treated as n additional implicit parameters; the argument list contains a total of $n+1$ items, with the extents following the main argument. The extents are always passed by the immediate mechanism.
3. OPTIONAL parameters are always passed by reference.
4. For a LIST parameter, the list mechanism is used. The items in the list are in accordance with conventions 1 and 2.
5. For OUT LIST parameters and IN OUT LIST parameters, Pillar uses the reference mechanism for the items in the list.

13.8.2 Linkage Specifications

A procedure-type-specification can contain a procedure-linkage-specification to explicitly control some aspects of the procedure call implementation.

■ procedure-linkage-specification

$$\text{LINKAGE} \left\{ \begin{array}{l} \text{IMMEDIATE} (\{ \text{identifier} \} , \dots) \\ \text{REFERENCE} (\{ \text{identifier} \} , \dots) \\ \text{DESCRIPTOR} (\{ \text{identifier} \} , \dots) \\ \text{TRAILING LIST} (\text{identifier}) \end{array} \right\} , \dots$$

If IMMEDIATE occurs, each identifier must be the name of one of the procedure type's parameters. The immediate mechanism will be used for the parameter. IMMEDIATE is only allowed on parameters that would normally be passed by the immediate mechanism.⁸

If REFERENCE occurs, each identifier must be the name of one of the procedure type's parameters. The reference mechanism will be used for the parameter.

⁷ The meaning of these system conventions needs to be defined for each system. It is probably the case that:

- Small OUT and IN OUT parameters are passed by the immediate mechanism on RISC architectures, but are passed by reference on VAX.
- Small IN parameters whose size is greater than 32 bits (this includes DOUBLE, QUADWORD, and LARGE_INTEGER) are passed by the immediate mechanism on RISC architectures, but are passed by reference on VAX.

⁸ The only purpose of IMMEDIATE, then, is to note the fact that this mechanism is being used. A possible future extension of the language is to allow it on types that would not ordinarily be passed immediately.

If **DESCRIPTOR** occurs, each identifier must be the name of one of the procedure type's parameters. The parameter's type must be one for which the system has a descriptor. The descriptor mechanism will be used for the parameter. The only types to which **DESCRIPTOR** applies are **STRING(*)**, **VARYING_STRING(*)**, and **BYTE_DATA(*)**.

It is an error if, in all of the identifier lists following **IMMEDIATE**, **REFERENCE** and **DESCRIPTOR**, a parameter is specified more than once. It is an error to specify the same parameter as both **DESCRIPTOR** and **TRAILING LIST**.

If **TRAILING LIST** is specified, the name must be the name of the last parameter in the procedure type's parameter list, and that parameter must be a **LIST** parameter. The list of argument items corresponding to the parameter will be passed as the last part of the argument list. **TRAILING LIST** is only allowed on a parameter whose position in the argument list is such that, without **LIST**, it would be passed in the memory argument list rather than in a register.⁹

⁹ Therefore, the parameters on which **TRAILING LIST** is allowed is system dependent.

CHAPTER 14

TARGET-SPECIFIC FEATURES

This chapter is not yet written.

APPENDIX A

COLLECTED SYNTAX

■ alerts-enable

DISABLE ALERTS ;

■ alignment-option

$$\text{ALIGNMENT} \left(\begin{array}{l} \text{BYTE} \\ \text{WORD} \\ \text{LONGWORD} \\ \text{QUADWORD} \end{array} \right)$$

■ alternative

$$\text{WHEN} \left\{ \begin{array}{l} \text{set-of-values} \\ \text{OTHERS} \end{array} \right\} \text{ THEN } \left\{ \begin{array}{l} \text{union-field-list} \\ \text{NOTHING ;} \end{array} \right\}$$

■ argument

expression

■ argument-list

$$\left[\begin{array}{l} \text{argument} \\ \text{keyword-argument} \end{array} \right], \dots$$

■ array-constructor

$$\text{"{"} \left\{ \begin{array}{l} \text{element-list} \\ \left\{ \begin{array}{l} \text{selected-element-value} \\ \text{selected-subarray-value} \end{array} \right\} \left[\text{ , OTHERS = initializer } \right] \end{array} \right\} \text{"}"$$

■ array-selector

[{ constant-expression } , ...]

■ array-type-constructor

ARRAY [{ range-specification } , ...] OF type-specification

■ assert-statement

ASSERT { expression [ELSE character-string-literal] } , ... ;

■ assignment-statement

data-reference = expression ;

■ binary-literal

" { binary-digit-string }... " { b
B }

■ bind-declaration

[BIND] identifier = data-reference ;

■ bound-flexible-type

name ({ * expression } ,...)

■ built-in-function-call-statement

built-in-function-invocation ;

■ built-in-function-invocation

name ([argument-list])

■ built-in-function-reference

builtin-function-invocation

■ case-statement

CASE expression
{ WHEN set-of-values THEN statement-sequence } ...
[WHEN OTHERS THEN statement-sequence]
END CASE ;

■ character-string-literal

" [[non-escaped-character
escaped-character]] ... "

■ comment

! [[non-escaped-character
tab
\
"]] ...

■ complete-condition-declaration

[CONDITION] identifier : condition-severity [= character-string-literal] ;

■ complete-message-declaration

[MESSAGE] identifier = character-string-literal ;

■ complete-opaque-type-declaration

[TYPE] identifier OPAQUE [size-option] : type-specification ;

■ complete-procedure-declaration

[PROCEDURE] identifier procedure-type-specification
**[[INLINE [ONLY]
WITH ARGUMENT PROBING]]** ;
local-block PROCEDURE identifier ;

■ complete-value-declaration

[VALUE] identifier : type-specification = initializer ;

■ complete-variable-declaration

[VARIABLE] { identifier } ,... : type-specification **[= initializer]** **[[ALIASED
SHARED]]** ;

■ component-list

{ identifier **[*]** } ,...

■ components-layout

{ field-component
filler-component } ...
union-layout

■ compound-statement

[WITH] local-block ;

■ condition-handler-enable

ENABLE CONDITION HANDLER name ;

■ condition-severity

{ SUCCESS
INFORMATIONAL
WARNING
ERROR
FATAL }

■ constant-expression

expression

■ data-reference

{ name
procedure-function-reference
built-in-function-reference
indirect-reference
dot-qualified-reference
indexed-element-reference
substring-reference
type-cast-reference }

■ decimal-literal

decimal-digit-string

■ declaration-completion

{ opaque-type-completion
value-completion
variable-completion
procedure-completion }

■ define-declaration

[DEFINE] identifier = data-reference ;

■ definition-module

MODULE identifier [module-identification] ;
[import-section]
{ general-module-level-declaration
complete-opaque-type-declaration } ...
external-declaration
[linkage-option-section]
END MODULE identifier;

■ dot-qualified-reference

{ data-reference.identifier
name.identifier }

■ element-list

{ { [constant-expression OF] initializer } , ... [, OTHERS = initializer] }
OTHERS = initializer }

■ enable-section

[lock-enable]
[alerts-enable]
[underflow-enable]
[condition-handler-enable]
[message-vector-enable]
[environment-enable]

■ enumerated-type-constructor

({ identifier } , ...) [QUALIFIED]

■ environment-declaration

ENVIRONMENT identifier [EXTENDS (name)] ;
[member-variable-declaration]...
END ENVIRONMENT ;

■ environment-enable

ENABLE ENVIRONMENT name ;

■ escaped-character

$$\left\{ \begin{array}{l} \backslash \backslash \\ \backslash " \\ \backslash (\text{decimal-digit-string}) \\ \backslash (\text{identifier}) \end{array} \right\}$$

■ exception-handler-section

WHEN exception-list THEN statement-sequence

■ exception-list

$$\left\{ \begin{array}{l} \text{exception-name-list} \\ \text{exception-others} \end{array} \right\}$$

■ exception-name-list

EXCEPTION { name } ,...

■ exception-others

EXCEPTION OTHERS

■ exit-loop-statement

EXIT LOOP ;

■ exponent

$$\left\{ \begin{array}{l} e \\ E \end{array} \right\} \left[\left[\begin{array}{l} + \\ - \end{array} \right] \right] \text{decimal-digit-string}$$

■ expression

$$\left\{ \begin{array}{l} \text{literal-constant} \\ \text{set-constructor} \\ \text{data-reference} \\ (\text{expression}) \\ \text{relational-expression} \\ \text{infix-operator-expression} \\ \text{prefix-operator-expression} \\ \text{array-constructor} \\ \text{record-constructor} \\ \text{NIL} \end{array} \right\}$$

■ extent-parameter-declaration-list

$$\left(\left\{ \left\{ \text{identifier} \right\} , \dots : \left\{ \begin{array}{l} \text{name} \\ \text{subrange-type-constructor} \end{array} \right\} \right\} ; \dots \left[\text{; 1} \right] \right)$$

■ external-condition-declaration

[CONDITION **]** { identifier } ,... EXTERNAL ;

■ external-declaration

{
external-value-declaration
external-variable-declaration
external-message-declaration
external-condition-declaration
external-procedure-declaration
external-opaque-type-declaration
}

■ external-message-declaration

[MESSAGE] { identifier } , ... EXTERNAL ;

■ external-opaque-type-declaration

[TYPE] identifier OPAQUE [size-option] EXTERNAL ;

■ external-procedure-declaration

[PROCEDURE] identifier procedure-type-specification
[INLINE [ONLY]] EXTERNAL ;

■ external-value-declaration

[VALUE] identifier : type-specification EXTERNAL ;

■ external-variable-declaration

[VARIABLE] { identifier } , ... : type-specification EXTERNAL [[ALIASED
SHARED]] ;

■ field-component

identifier , ... [alignment-option] [position-option] ;

■ field-declaration

{ identifier } , ... : type-specification [= initializer] ;

■ field-list

[[field-declaration] ...
union
[variant-part]

■ filler-component

identifier : FILLER ({
BIT
BYTE
WORD
LONGWORD
QUADWORD
} , { constant-expression })

■ flexible-type-declaration

[TYPE] identifier(extent-parameter-declaration-list) :

{
named-type
set-type-constructor
array-type-constructor
record-type-constructor
}

■ floating-point-literal

{
decimal-digit-string exponent
decimal-digit-string . decimal-digit-string [exponent]
.decimal-digit-string [exponent]
}

■ general-control

FOR identifier [: named-type] = expression
NEXT expression

■ general-module-level-declaration

{
constant-declaration
normal-type-declaration
flexible-type-declaration
procedure-type-declaration
complete-value-declaration
complete-variable-declaration
environment-declaration
define-declaration
complete-message-declaration
complete-condition-declaration
complete-procedure-declaration
external-declaration
}

■ global-synonym-option

{name = character-string-literal} ,... ;

■ goto-statement

GOTO identifier ;

■ hexadecimal-literal

" { hexadecimal-digit-string }... " { X
X }

■ identifier

{ letter
\$
- } [[letter
\$
-
decimal-digit]] ...

■ if-statement

```
IF expression THEN statement-sequence  
  [ ELSEIF expression THEN statement-sequence ] ...  
  [ ELSE statement-sequence ]  
END IF ;
```

■ implement-section

```
{ IMPLEMENT { module-implement } ... } ...
```

■ implementation-module

```
MODULE identifier [ module-identification ] ;  
  { implement-section } ...  
  [ import-section ]  
  { general-module-level-declaration  
    declaration-completion } ...  
  [ linkage-option-section ]  
END MODULE identifier;
```

■ import-section

```
[ IMPORT { module-import } ... ] ...  
[ REVEAL { name } , ... ; ]
```

■ increment-or-decrement-control

```
FOR identifier [ : named-type ] = expression  
  [ BY expression ] [ DOWN ] TO expression
```

■ indexed-element-reference

```
{ data-reference [ { expression } , ... ]  
  name [ expression ] }
```

■ indirect-reference

```
data-reference^
```

■ infix-operator-expression

```
subexpression { *  
                **  
                +  
                -  
                /  
                (+)  
                AND  
                OR  
                XOR  
                MOD } subexpression
```

■ initializer

```
expression
```

■ keyword-argument

identifier = { argument } , ...

■ layout-list

[components-layout] ...
[variant-part-layout]

■ line

[F] [white-space] [token [white-space]]... [comment]

■ linkage-option-section

LINKAGE OPTIONS
[qualified-globals-option]
[global-synonym-option]
[linker-value-option]

■ linker-value-option

LINKER VALUE ({ name } ,...) ;

■ literal-constant

{ decimal-literal
binary-literal
octal-literal
hexadecimal-literal
floating-point-literal
character-string-literal } [: named-type]

■ local-block

[{ local-block-declaration-section } ...]
[enable-section]
BEGIN
statement-sequence
[exception-handler-section] ...
[unwind-handler-section]
[subprocedure-section]
END

■ local-block-declaration-section

{ normal-type-declaration
flexible-type-declaration
complete-opaque-type-declaration
procedure-type-declaration
constant-declaration
complete-value-declaration
complete-variable-declaration
bind-declaration
define-declaration
complete-message-declaration
complete-condition-declaration }

■ lock-enable

LOCK data-reference ;

■ loop-statement

$$\left[\begin{array}{l} \text{ordinal-type-control} \\ \text{increment-or-decrement-control} \\ \text{general-control} \end{array} \right] \text{ [WHILE expression]}$$

LOOP
statement-sequence
END LOOP [identifier] ;

■ member-variable-declaration

{ identifier } , ... : type-specification ;

■ message-vector-enable

ENABLE MESSAGE VECTOR name ;

■ module

$$\left\{ \begin{array}{l} \text{program-module} \\ \text{definition-module} \\ \text{implementation-module} \end{array} \right\}$$

■ module-identification

IDENTIFICATION (character-string-literal)

■ module-implement

identifier COMPONENTS component-list ;

■ module-import

identifier [COMPONENTS component-list] ;

■ name

$$\left\{ \begin{array}{l} \text{unqualified-name} \\ \text{qualified-name} \\ \text{reserved-name} \end{array} \right\}$$

■ named-type

$$\left\{ \begin{array}{l} \text{[POINTER] name} \\ \text{bound-flexible-type} \end{array} \right\}$$

■ normal-type-declaration

[TYPE] identifier :
$$\left\{ \begin{array}{l} \text{type-specification} \\ \text{enumerated-type-constructor} \end{array} \right\} ;$$

■ nothing-statement

NOTHING ;

■ octal-literal

" { octal-digit-string }... " { ^o / _o }

■ opaque-type-completion

[TYPE] identifier : type-specification ;

■ ordinal-type-control

FOR identifier IN named-type

■ parameter-declaration

[[IN
OUT
IN OUT
BIND]] { identifier } , ... : { named-type
subrange-type-constructor
set-type-constructor } parameter-options ;

■ parameter-default-value

= initializer

■ parameter-list

([parameter-declaration] ...)

■ parameter-options

[CONFORM] [[parameter-default-value
OPTIONAL
LIST [[range-specification]]]] [KEYWORD] [STATUS VECTOR]

■ parameter-repetition

([identifier] , ...) [RETURNS [identifier]]

■ position-option

POSITION ({ BIT
BYTE
WORD
LONGWORD
QUADWORD } , constant-expression)

■ prefix-operator-expression

[[+
-
NOT]] { literal-constant
data-reference
(subexpression) }

■ procedure-call-statement

procedure-invocation ;

■ procedure-completion

[PROCEDURE] identifier parameter-repetition ; [WITH ARGUMENT PROBING ;] local-block
PROCEDURE identifier ;

■ procedure-function-reference

procedure-invocation

■ procedure-invocation

data-reference ([argument-list])

■ procedure-linkage-specification

LINKAGE { IMMEDIATE ({ identifier } , ...)
REFERENCE ({ identifier } , ...)
DESCRIPTOR ({ identifier } , ...)
TRAILING LIST (identifier) } , ...

■ procedure-result-specification

RETURNS [identifier :] { named-type
subrange-type-constructor
set-type-constructor }

■ procedure-type-constructor

parameter-list [procedure-result-specification]

[ENVIRONMENT (name)]

[procedure-linkage-specification]

■ procedure-type-declaration

[TYPE] identifier :

PROCEDURE procedure-type-specification

■ procedure-type-reference

parameter-repetition OF TYPE name

■ procedure-type-specification

{ procedure-type-constructor
procedure-type-reference }

■ program-module

PROGRAM identifier ENTRY name [module-identification] ;

[import-section]

{ general-module-level-declaration }...

[linkage-option-section]

END PROGRAM identifier;

■ qualified-globals-option

QUALIFIED GLOBALS;

■ qualified-name

name . identifier

■ raise-statement

$$\text{RAISE } \left\{ \begin{array}{l} \text{ERROR [character-string-literal]} \\ \left\{ \begin{array}{l} \text{EXCEPTION} \\ \text{REPORT} \end{array} \right\} \text{data-reference [argument] , ...} \\ \text{VECTOR data-reference} \end{array} \right\} ;$$

■ range-specification

$$\left\{ \begin{array}{l} \text{[expression] .. [expression]} \\ \text{expression : (expression)} \\ \text{named-type} \end{array} \right\}$$

■ record-constructor

$$\text{"{" } \left[\left[\left\{ \text{identifier = initializer} \right\} \dots \left[\text{ , OTHERS = DEFAULT } \right] \right] \right] \text{"}$$

■ record-layout-option

$$\text{LAYOUT [size-option]} \\ \left\{ \begin{array}{l} \text{layout-list} \\ \text{PACKED IN ORDER} \\ \text{ALIGNED IN ORDER} \end{array} \right\} \\ \text{END LAYOUT ;}$$

■ record-type-constructor

$$\text{RECORD [CAPTURE EXTENTS ;] [EXTENSIBLE ;]} \\ \left[\text{EXTENDS named-type ;} \right] \\ \left\{ \begin{array}{l} \text{field-list} \\ \text{NOTHING;} \end{array} \right\} \\ \left[\text{record-layout-option} \right] \\ \text{END RECORD ;}$$

■ relational-expression

$$\text{subexpression } \left\{ \begin{array}{l} < \\ <= \\ == \\ <> \\ >= \\ > \end{array} \right\} \text{subexpression}$$

■ reserved-name

identifier

■ return-statement

RETURN [expression] ;

■ selected-element-value

array-selector = initializer

■ selected-subarray-value

array-selector = "{" element-list "}"

■ set-constructor

[[set-of-values]] [: named-type]

■ set-of-values

{ expression
range-specification } ...

■ set-type-constructor

{ SET name [range-specification] [size-option]
name size-option }

■ simple-expression

expression

■ size-option

SIZE ({ BIT
BYTE
WORD
LONGWORD
QUADWORD } [[, simple-expression]])

■ statement

{ assert-statement
assignment-statement
built-in-function-call-statement
case-statement
compound-statement
exit-loop-statement
goto-statement
if-statement
loop-statement
nothing-statement
procedure-call-statement
raise-statement
return-statement }

■ statement-sequence

{ [identifier :] statement } ...

■ subexpression

expression

■ subprocedure-section

SUBPROCEDURES { complete-procedure-declaration } ...

■ subrange-type-constructor

{ name [range-specification] [size-option]
name size-option }

■ substring-reference

data-reference [range-specification]

■ token

{ identifier
compile-time-facility-symbol
decimal-literal
binary-literal
octal-literal
hexadecimal-literal
floating-point-literal
character-string-literal
punctuation-symbol }

■ type-cast-reference

data-reference :: { named-type [TRUNCATE] }

■ type-specification

{ named-type
subrange-type-constructor
set-type-constructor
array-type-constructor
record-type-constructor }

■ underflow-enable

{ ENABLE
DISABLE } UNDERFLOW ;

■ union

UNION CASE { identifier
* }
{ alternative } ...
END UNION ;

■ union-field-list

{ field-declaration } ...
union }

■ union-layout

UNION [alignment-option] [position-option]
{ OVERLAY components-layout } ...
END UNION ;

■ unqualified-name

identifier

■ unwind-handler-section

WHEN UNWINDING THEN statement-sequence

■ value-completion

[VALUE] identifier = initializer ;

■ variable-completion

[VARIABLE] identifier [= initializer] ;

■ variant

WHEN { set-of-values
OTHERS } THEN { field-list
NOTHING ; }

■ variant-part

VARIANTS CASE identifier
{ variant } ...
END VARIANTS;

■ variant-part-layout

VARIANTS [alignment-option] [position-option]
{OVERLAY layout-list}...
END VARIANTS;

■ white-space

{ space
tab } ...

GLOSSARY

actual type: That type which is hidden by an opaque type. It is the actual type that is made visible when the opaque type is revealed.

argument item: A member of the machine argument list. It is an item passed from the call site to the called routine, to represent the argument that is to be associated with a parameter in the called procedure.

arithmetic type: An integer type or floating-point type, all of which are legal in arithmetic operations.

assignable data item: A data item that can take on different values during the course of its existence.

assignable data reference: A reference to an assignable data item.

associated type: That type to which a pointer type points; that is, `POINTER t`'s associated type is *t*.

base ordinal type: 1. A primitive ordinal type or an enumerated type. 2. That type from which an ordinal type is derived.

BIND item: That which is declared by a bind-declaration.

bit-class type: A nonaddressable type that occupies *n* bits, and is usable in declarations only as the type of a record field or array element.

complete type: A tuple consisting of a type and a value for each of the type's extent parameters and free extents.

constant: A named constant or literal.

constant expression: An expression that is reducible to a constant by the rules of Pillar.

constant range: A range whose specification contains no nonconstant expressions.

constant type: A type that does not depend on nonconstant expressions.

data item: An object that can take on values. Some data items can occupy storage (and hence have locations). Such data items can be assignable or nonassignable.

DEFINE item: That which is declared by a define-declaration.

element: 1. An individual item contained in an array, `blank_DATA`, or string type is an element of that type. 2. Of a set type, one of the members that can be contained in that set type. 3. Of an enumerated type, one of the named constants declared by that enumerated type.

empty range: A range that contains no values.

exception: An event that happens during interpretation of a language construction, and terminates the interpretation.

extension: If record type *r* extends record type *t*, that part of *r* not contained in *t*.

extent parameter: One of the parameters of a flexible type.

flexible type: A type declared with parameters that defines a family of types.

floating-point type: One of the types REAL or DOUBLE.

free extent: A nonconstant value, other than an extent parameter, on which a type depends.

index range: One of the ranges in the definition of an array type that defines values that a subscript can assume.

inline procedure: A procedure declared as `INLINE ONLY` or `INLINE`, that the compiler must or might expand using inline code, rather than a call.

integer type: A type having `INTEGER` or `LARGE_INTEGER` as its base ordinal type.

local-block: A compound-statement or the body of a procedure (including any subprocedures).

main statement-sequence: The statement-sequence immediately contained in a local-block.

matching extent: An extent value given as an asterisk; its value is derived from the type of another item.

module: The unit of a Pillar compilation.

name: An identifier associated with an object.

named constant: A nonassignable data item declared by a `CONSTANT` declaration, or declared as an element of an enumerated type.

named-type: A type that can be referred to by name, or a pointer to such a type, or (if the type is flexible) a binding of such a type.

nonassignable data item: A data item whose value is fixed during the course of its existence.

nonassignable data reference: A reference to a nonassignable data item.

primitive type: One of the types that is part of the Pillar language, and does not have to be declared by a user.

procedure activation: A run-time instance of a procedure being called.

range: A contiguous set of ordinal values delineated by a low and high value.

reference: A syntactic construction used to refer to a data item.

root type: That type to which a type can be reduced no further; used for the purpose of defining type sameness and equivalence.

Usage: This is similar to what used to be called a base type.

scope: The module, modules, or parts of a module in which a declaration is known.

selector: The value that determines which alternative or variant is currently active (selected).

small type: A specified set of types, data items of which, in general, can be contained in registers, and can be operated on inexpensively.

target type: That type which, in a given context, specifies the type to which an expression must be converted, if the expression is not already of that type.

type: A property of a data item that specifies a set of values to be assumed by the item, and a set of operations allowed on the item.

variable: An assignable data item declared by a **VARIABLE** declaration.

Index

A

Array-constructor, 68
Array-selector, 70
Assert-statement, 114
ASSERT_CHECK_ENABLED built-in constant,
115
Assignment-statement, 115
Atomic operation
 READ/WRITE_REGISTER, 121

B

Binary-literal, 11
Built-in function
 calling as statement, 120
Built-in-function-call-statement, 120
BY
 in loop, 119

C

Case-statement, 115
Character set, 5
Character-string-literal, 12
Comment, 10
Compilation unit, 21
Component-list, 24
Compound-statement, 116
Constant
 literal, 65
 constants, 65
Constructor, 67

D

Decimal-literal, 11
Declaration
 EXTERNAL, 27
Default
 as initializer, 70
 type of control variable, 119
DOWN TO
 in loop, 119

E

Element-list, 69
ELSE, 117
ELSEIF, 117
END statement
 executing in procedure, 124
Escaped-character, 12
EXIT LOOP statement, 116
Exponent, 12
External declaration, 27

F

Floating-point-literal, 12
FOR loop, 118, 119, 120

G

General-control
 of loop, 120
GOTO-statement, 116

H

Hexadecimal-literal, 11

I

Identifier, 9, 11
IF statement, 117
Implement-section, 26
Import-section, 24
%INCLUDE, 13
Increment-or-decrement-control
 of loop, 119
Initializer, 67
 array, 68
 record, 70

K

Keyword, 9

L

Label
 allowed as target of GOTO, 116
 declaration of, 114
Line, 10
Linkage-option-section, 29
Linker value option, 31
Literal
 character string, 12
 floating point, 12
 numeric, 11
Literal-constant, 65
Loop
 EXIT LOOP, 116
 LOOP statement, 117

M

Module
 defined, 21

N

Named-type
 in FOR loop, 119
Nested declaration, 17
NEXT
 in loop, 120
NIL
 as initializer, 67
Nonlocal GOTO, 117
Nonsequential control flow, 113
NOTHING statement, 120

O

Octal-literal, 11
Ordinal-type-control
 of loop, 118
Others
 in array constructor, 69
OTHERS
 in CASE statement, 115
 in record constructor, 70

P

PILLAR\$_ASSERT, 115
PILLAR\$_ERROR, 123
Procedure-call-statement, 122
Punctuation, 13

R

RAISE statement, 123
Range violation
 failed assertion, 114
 in CASE statement, 116
 in continuation of loop, 119
 in ordinal-type-control, 118
 in READ_REGISTER, 121
 in return-statement, 124
 in statement-sequence, 113
 in WRITE_REGISTER, 121
 loop increment value, 119
 loop limit value, 119
READ_REGISTER built-in function, 121
Record-constructor, 70
Record field
 with WRITE_REGISTER, 122
Reserved word, 9
Result-variable, 124
RETURNS option, 124
RETURN statement, 124
Row major
 array initializer, 69

S

Scope
 defined, 17
 nested, 17
Selected-element-value, 69
Selected-subarray-value, 70
Selector expression
 in CASE statement, 116
Set
 literal source, 66
Set constructor, 67
set-of-values
 in CASE statement, 116
Set-of-values, 68
Source line, 8, 10
Source module, 21
 characters, 5
 defined, 8
Statement, 114
Statement-sequence, 113
Subarray
 in array constructor, 69
Subrange
 field initializer, caution, 71

T

TO
 in loop, 119
token, 10
Token
 defined, 9

U

Union

in record constructor, 70

V

Variant

in record constructor, 70

W

WHEN

in CASE statement, 115

WHEN clause

in CASE statement, 116

values, 68

WHILE loop, 118

White-space, 10

Wildcard

in importing, 25

WITH

in compound statement, 116

WRITE_REGISTER built-in function, 121