MHDL Language Reference Manual

IR-VA-054

Version 2.3

February 21, 1996

Contract No. DAAL01-93-C-3368

Prepared by:

Intermetrics, Inc.
7918 Jones Branch Drive
McLean, Virginia 22102

Prepared for:

ARPA/Tri-Service MHDL Initiative

Published by
Intermetrics, Inc.
7918 Jones Branch Drive, McLean, Virginia 22102

Copyright ©1996 by Intermetrics, Inc.

This document is furnished with unlimited rights as set forth in Defense Federal Acquisition Regulation Supplement (DFARS) clause 252.227-7013, unless otherwise noted. This material may be reproduced by or for the United States Government pursuant to the copyright license under the clause at DFARS 252.227-7013 (Oct 1988).

Additionally, users of this documentation are free to make copies of the documentation for the IEEE SCC-30 MHDL Study Group usage, provided that such copies are not resold or otherwise distributed, in any form, to third parties by these users. This copyright notice and legend must be included on all copies made.

This work is sponsored by the ARPA/Tri-Service MHDL Initiative under contract DAAL01-93-C-3368.

Contents

1 Introduction **1**

1.1 Notational Conventions 2

2 Models **5**

2.1 Inclusion Lists 7

2.2 Packages 9

2.3 Generate Definitions 9

2.4 Tables 13

2.5 Duplicate Definitions 15

2.6 Libraries 15

3 Structure **17**

3.1 Structure Definitions 17

3.1.1 Components 19

3.1.2 Connectors 21

3.1.3 Connection List 21

3.2 Configurations 22

3.3 Design Hierarchies 25

3.4 Elaboration 26

3.5 Layouts 28

4 Types and Classes **29**

4.1 Type Definitions 30

4.1.1 Specified Types 30
4.1.2 Primitive Types 36
4.1.3 Type Classes 37
4.1.4 Class Instances 39
4.2 Predefined Types 40
4.2.1 Booleans 41
4.2.2 Characters and Strings 41
4.2.3 Function Types 41
4.2.4 Lists 41
4.2.5 Tuples 42
4.2.6 Arrays 42
4.2.7 Null Type 42
4.2.8 Integers 42
4.2.9 Rationals 43
4.2.10 Reals 43
4.2.11 Complex Numbers 43
4.2.12 Physical Types 43
4.2.13 Signal Type 50
4.3 Type Coercions 51

5 Properties 55
5.1 Abstract Properties 55
5.2 Property Value Definitions 57
5.3 Associations 58

6 Named Values 61
6.1 Name Definitions 62
6.2 Value Definitions 63
6.2.1 Function Bindings 65
6.2.2 Pattern Bindings 69
6.3 Patterns and Pattern Matching 72

6.4 Constraints 75
6.5 Equations 76

7 Expressions **79**
7.1 Function Applications 81
7.2 Operator Applications 82
7.3 Lambda Abstractions 83
7.4 Let Expressions 84
7.5 Conditional Expressions 85
7.6 Case Expressions 85
7.7 Typed Expressions 87
7.8 Structure Function Creations 88
7.9 Derivatives and Integrals 89
7.10 Simple Expressions 91
7.10.1 Literals 91
7.10.2 Null Expressions 92
7.10.3 Parenthesized Expressions 92
7.10.4 Tuples 92
7.10.5 Lists 93
7.10.6 Arithmetic Sequences 93
7.10.7 Generators 94
7.10.8 Physical Values 95
7.10.9 Primitive Values 96

8 Names **99**
8.1 Simple Names 100
8.2 Record Names 100
8.3 Qualified Names 101
8.4 Hierarchical Names 102
8.5 Special Names 103
8.6 Visibility 104

9 Lexical Rules **107**

9.1 Character Set 107

9.2 Lexical Elements and Separators 107

9.3 Identifiers 108

9.3.1 Basic Identifiers 108

9.3.2 Escaped Identifiers 109

9.4 Reserved Words 110

9.5 Literals 110

9.5.1 Integer and Floating-Point Literals 111

9.5.2 Character and String Literals 113

9.6 Operator Symbols and Reserved Operators 115

9.7 Data Tables 115

9.8 Comments 116

10 Standard Definitions **119**

10.1 Numeric Types 120

10.1.1 Eq 120

10.1.2 Ord 120

10.1.3 Group 121

10.1.4 Num 121

10.1.5 Mult 121

10.1.6 Ix 122

10.1.7 Enum 122

10.1.8 Integers 122

10.1.9 Fractional 123

10.1.10 Real 123

10.1.11 RealFrac 124

10.1.12 RealFloat 125

10.1.13 Int 125

10.1.14 Ratio 126

10.1.15 Float 127

10.1.16 Complex 128
10.1.17 Physical 129
10.2 Standard Enumerations 138
10.2.1 Bool 138
10.2.2 Char 139
10.2.3 Text 141
10.2.4 Binary 145
10.3 Composite Types 148
10.3.1 Monad 148
10.3.2 List 149
10.3.3 Array 156
10.3.4 Exceptions 161
10.3.5 Topology 162
10.4 Signal Types 163
10.4.1 Universal Signals 163
10.4.2 Continuous Signals 166
10.4.3 Discrete Signals 167
10.4.4 Periodic Signals 167
10.5 Standard Functions 168
10.5.1 Function Class 168
10.5.2 Coercion Functions 169
10.5.3 Misc. Functions 170
10.5.4 Structural Functions 173
10.5.5 Numeric Functions 176
10.5.6 Random Functions 177
10.6 MHDL Standard 177
10.6.1 VHDL 177
10.6.2 Connector Types 179
10.6.3 Ground 182

A Syntax Summary **187**

B Glossary **207**

Combined Index **221**

Chapter 1

1 Introduction

2 This is the latest release (Version 2.3) of the Language Reference Manual (*LRM*) of the MIMIC Hardware Description Language (*MHDL*). The language is largely complete.

3 The language is intended to represent analog and microwave designs and hardware. It includes language features for describing the physical connections of the device, the hierarchical decomposition of the electronic design, and the functional behavior of the system and its components.

4 This revision reflects extensive comments concerning the language, and in particular a number of syntax changes on the part of the MHDL Working Group. There are a number of clarifications and rephrasings of different parts of the language, and a number of language issues have been closed out as a result of this revision.

5 The format of the manual has been changed. Paragraphs are no longer indented, and are separated by white space; this is done for readability. Paragraph numbers have been added, and change bars. The change bars reflect changes against the February 7, 1995 version of the LRM (Version 2.1).

6 **Note:**

> Please submit comments to the electronic address i2mhdl@wash.inmet.com if they do not require discussion by the broader MHDL community. Issues that do require such discussion, or that should be seen by a wider audience, may be submitted to the standard MHDL mailing list at mhdl@nosc.mil.

7 This document is in response to the MHDL Requirements Document ([8]). The language, as given in this document, satisfies almost all of these requirements, with the

exception of some of the standard definitions. These await further analysis on the part of the MHDL Working Group.

8 The overall structure of the document starts with the basic encapsulation construct: models and packages. From there, the basic features of the language are covered in order of increasing complexity: structure, types and classes, properties, functions, expressions, names, lexical rules, and standard definitions. An extensive index is provided.

9 In addition to soliciting as much comment on the language structures and features as possible, the authors of this document are explicitly requesting comments on the document format itself. This document is provided in two forms:

- As a text document.
- As a file for an electronic browsing system (Info).

10 Two complaints concerning past reference manuals were the lack of an index and the lack of electronic cross-reference. Both of those lacks are being addressed with this document format. We are requesting comments concerning additional index entries and additional cross-references and menu items.

11 This document contains examples in the form of brief code fragments at the end of appropriate sections. The examples are not a part of the language definition; they are included in order to provide the reader with a very general idea of the way in which various language constructs might be used.

1.1 Notational Conventions

1 The MHDL syntax is defined by a context-free grammar specified in a meta-language that is a variant of the Backus-Naur Form (BNF). The grammar consists of a set of rules that describe the form of each MHDL syntactic construct (or syntactic category). Each rule has the following form:

2
```
syntactic_category ::= string_of_symbols
```

3 The meta-symbol "::=" indicates that the rule defines the syntactic category to the left of the symbol, and the string of symbols to the right is a sequence of terminal and nonterminal symbols defining the structure of the syntactic construct. A nonterminal symbol is one that can be further decomposed into other symbols; this decomposition is specified by another grammar rule. A terminal symbol is literal text that cannot be further decomposed.

4 The following rules give a precise definition of the notation used to describe the MHDL syntax:

1. Syntactic categories are represented by lower case words. These words may contain underscore characters, but not blanks. For example:

   ```
   generate_definition
   ```

   The name of a syntactic category may begin with an upper case part providing additional semantic information; the name of such a syntactic category is equivalent to the syntactic category name without the upper case part. For example,

   ```
   BOOLEAN_expression
   ```

   is equivalent to

   ```
   expression
   ```

   In the text of the LRM, syntactic categories are identified by using the category name, without underscores, and without special fonts. Such syntactic categories are defined by their appearance in the grammar, and no further definition is given in the text.

2. Reserved words are represented in italics. For example:

   default

3. Rules are separated by blank lines; blank lines do not appear within rules. For example, the following are two separate rules:

   ```
   function_definitions ::= { function_definition }

   variables ::= variable { , variable }
   ```

4. Alternatives are separated by vertical bars. The vertical bar binds as loosely as possible; i.e., it applies to the largest possible expressions on either side. For example:

   ```
   general_type ::= parameterized_type | simple_type
   ```

5. Optional items are enclosed by square brackets. Hence, the following rules are equivalent:

   ```
   exposed_item ::= variable [ -> variable ]

   exposed_item ::= variable | variable -> variable
   ```

6. Repeated items are enclosed by curly braces. The item may appear zero or more times, and repetitions are expanded from left to right. For example, the rule

   ```
   type_specification ::= TYPE_identifier { TYPE_variable }
   ```

   specifies that a type specification consists of an identifier followed by zero or more type variables.

7. The characters `'|'`, `'{'`, `'}'`, `'['`, `']'` are enclosed in single quotes when they stand for themselves (i.e. when they are used as terminal symbols rather than meta-symbols). In the following example, `'['` and `']'` are terminal symbol rather than a meta-symbol:

```
list_type ::= '[' type_expression ']'
```

Chapter 2

Models

Models allow definitions to be grouped together to form a single unit.

Example:

```
model Amplifier

  attribute Gain : Potential;
  default Gain = 1.0 'volts';

end Amplifier;
```

Grammar Rules:

```
model ::=
    model MODEL_simple_name
        [ library_list ]
        [ inclusion_list ]
        [ model_body ]
    end [ MODEL_simple_name ] ;

model_body ::=
    model_item ; { model_item ; }

model_item ::=
    structure_definition
```

```
| structure_item
| class_definition
| instance_definition
| type_definition
| dimension_definition
| unit_definition
| multiplier_specification
```

3 The *model* is the basic unit of structure in MHDL. A model corresponds to any part of a design which may be regarded as a separate conceptual unit, according to the best judgment of the designer. In general, any part of the design which may appear as a box on a block diagram or which might have an independent name should be encapsulated in a model.

4 A model consists of a named area that may contain any other definition (except another model or package). The sole purpose of a model is to group conceptually related definitions together; there is no automatic name hiding at model boundaries. Declarations within a model are ordered in the textual order in which they occur; however, there are no rules concerning the order of declaration and the use of types or attributes (see 4.1 [Type Definitions], page 30). This is important for function definitions (see 6.2.1.1 [Functions], page 67).

5 A model may be built from other models; in this case, we say that the model is *derived from* the other models. A model (called the derived model) is derived from other models (called the named models) by including the definitions of the models from which it is derived. This inclusion of definitions is accomplished by an *inclusion list* (see 2.1 [Inclusion Lists], page 7 for more details). A model may also affect the visibility of the definitions in the models from which it is derived (see 8.6 [Visibility], page 104).

6 The *inclusion list* of the model specifies a number of packages and models, possibly from a number of libraries. If the name of a package or model appears in the inclusion list of a model, it is as if the declarations in the package or model also appeared in the model (with the exception of the insertion of coercion functions; see 4.3 [Type Coercions], page 51, for more details).

7 While naming a package or model, the inclusion list may also control the visibility of the derived model over the definitions in the named model. It does this by directly specifying the names in the named model that may be either hidden or seen from the derived model. See 2.1 [Inclusion Lists], page 7, for more information.

8 If a package is named in the inclusion list of a package or model, then the including package or model is said to *inherit* the definitions in the named package; this process is called *inheritance*.

9 If a model name appears after the keyword *end*, it must be the same name that appears

after the keyword *model* at the beginning of the model.

2.1 Inclusion Lists

1 Inclusion lists control the use of the definitions of a description unit in other description units.

Example:

```
model op_amp includes voltage, amperage from amplifier;

...

end op_amp;
```

Grammar Rules:

```
inclusion_list ::=
    includes visibility_item ;
        { includes visibility_item ; }

visibility_item ::=
    exposure_clause
  | hiding_clause

exposure_clause ::=
    [ exposure_list from ] MODEL_OR_PACKAGE_name

hiding_clause ::=
    MODEL_OR_PACKAGE_name [ except hiding_list ]

exposure_list ::=
    exposure_item { , exposure_item }

hiding_list ::=
    DEFINITION_name { , DEFINITION_name }

exposure_item ::=
    DEFINITION_name [ as DEFINITION_simple_name ]
```

2 An inclusion list names the models from which a given model is derived and the packages inherited by a model or package. The effect of the inclusion list is as if the definitions in the models and packages mentioned in the inclusion list had been included in the model or package containing the inclusion list (with the exception of the insertion of coercion functions; see 4.3 [Type Coercions], page 51, for more details). This inclusion takes place in the appropriate definition list: that of the model, package, structure, or component associated with the site of the inclusion list. The order in which the models and packages are listed in the inclusion list is not significant.

3 It is an error if a package includes a model; similarly, it is an error if a structure or a component includes a model. It is also an error if the inclusion process creates an illegal definition list; for example, if a package included in a structure contains a class definition.

4 An inclusion list creates a partial order called "precedes" between the definitions in the description unit in the inclusion list (the included definitions) and the definitions in the description unit that contains the inclusion list (the native definitions). Each included definition precedes each native definition. Because precedes is a partial order, this relationship is transitive.

5 If the effect of an inclusion list in a structure or a component is to duplicate definitions that are included as a result of an inclusion list in the model or structure (respectively) that contains the structure or component, then the effect is to not include the definitions again in the inner textual scope (the structure or component, respectively). The definitions in the outer scope (the model or structure) are used.

6 The inclusion list may also control the visibility of the names in the various models. Names included in a hiding list are explicitly excluded from visibility. If an exposure list is included, the names in the list (called the *exposed names*) are included in the visibility map, and all others are excluded from the visibility map. For each name in the exposure list that includes the form "*as* name" following it, the visibility map is changed as follows. Each occurrence of the first name in the exposure item (called the replaced name) is replaced by the second name in the exposure map (the exposed name). It is an error if any name in a hiding list or an exposure list refers to a unit or a multiplier. See 8.6 [Visibility], page 104, for more details concerning visibility and visibility maps.

7 **Note:**

> Including different packages and models in the inclusion list may have the effect of including different declarations twice. This effectively has no effect. See 2.5 [Duplicate Definitions], page 15, for the rules concerning duplicate definitions in a model.

2.2 Packages

1 A package is a mechanism for grouping together related definitions used in many places.

Example:

```
package Deadband

  attribute Low_threshold, Hi_threshold :  Float;
  attribute Gain : Float;

end Deadband;
```

Grammar Rules:

```
package ::=
    package PACKAGE_simple_name
        [ library_list ]
        [ inclusion_list ]
        [ model_body ]
    end [ PACKAGE_simple_name ] ;
```

2 A *package* is a collection of definitions that may be included in a number of models. Packages are used to avoid duplication, and to assure type consistency across a number of models. When a package is named in an inclusion list, it is as if the definitions in the package were included in the model or package that inherits it (with the exception of the insertion of coercion functions; see 4.3 [Type Coercions], page 51, for more information).

3 If a package name occurs after the keyword *end*, it must be the same name as that after the keyword *package* at the beginning of the package.

2.3 Generate Definitions

1 A generate definition allows the controlled replication of a series of definitions.

Example:

```
for each component C in Electrical generate
```

```
    C.component_type = conservative_model;
  end generate;
```

Grammar Rules:

```
generate_definition ::=
     if_generate_definition
  | for_generate_definition
  | table_generate_definition

if_generate_definition ::=
    if_generate_header
      { else_if_generate }
      [ else_generate ]
    end generate

if_generate_header ::=
    if BOOLEAN_expression
      generate  generate_body

else_if_generate ::=
    else if BOOLEAN_expression
      generate generate_body

else_generate ::=
    else generate
      generate_body

for_generate_definition ::=
    for_generate_header
      generate_body
    end generate

for_generate_header ::=
    for each generate_object_kind
      GENERATE_simple_name
      [ in generate_list ]
      generate

generate_object_kind ::=
    component
```

```
    | connector
    | structure
    | member
    | connection set

generate_list ::=
    STRUCTURE_OR_MODEL_OR_COMPONENT_OR_PACKAGE_name
  | LIST_expression

table_generate_definition ::=
    table_generate_header
      generate_body
    end generate

table_generate_header ::=
    for each row
    [ GENERATE_simple_name ] in
    table TABLE_name generate

generate_body ::=
    generate_item ; { generate_item ; }

generate_item ::=
    structure_item
  | structure_definition
```

2 A *generate definition* is a mechanism for repeating a specified group of definitions for each object in an MHDL construct or for each member in a list. Generate definitions are used for writing general sets of definitions which apply to each element of some description; for example, to each component in a structure or to each member in a list or table.

3 The definitions between the first *generate* keyword and the *end* keyword (called the *generated definitions*) are replicated as follows. In the case of an if generate definition, the definitions in the generate body of the if generate header are replicated once if the expression is true; similarly, if the expression is false, the definitions in the generate body of the first else if generate whose expression is true (if any) are replicated once; if no such else if generate exists, then the definitions in the generate body of the else generate are replicated once (if it exists). All other generate bodies in the if generate definition are replicated zero times. In the case of a table generate definition, the generated definitions are replicated once for each row in the table identified by the

table name. In the case of a for generate definition, the definitions are replicated once for each element indicated by the for generate header, as follows.

4 If the generate object kind is anything except *member*, the generated definitions are replicated for each occurrence of the object in the design hierarchy indicated by the generate object kind (component, structure, connector, or connection set) in its *definition set*. If a generate list occurs, the definition set is the definitions which are specified by the generate object kind in the component, structure, or model definition whose name occurs in the generate list (a list expression is illegal in this case). If no generate list occurs, the definition set is the definitions specified by the generate object kind that occurs in the containing structure if the generate definition occurs within a structure; otherwise, the definition set is the definitions specified by the generate object kind in the part that contains the generate definition.

5 If the generate object kind is *component* or *connector*, the generate list must be a structure name. If the generate object kind is *structure*, the generate list must be a model name or package name. If the generate object kind is *connection set*, the generate list must be a structure name (a connection set is part of a structure if it contains one of the connectors of that structure).

6 If the generate object kind is *member*, a generate list consisting of a list expression must be present. The generated definitions are replicated once for each member in the list. The list must not be infinite.

7 In the case of a for generate header, each time the generated definitions are replicated, all occurrences of the generate name in those definitions are replaced by the name of the corresponding definition (component, structure, connector, or connection set) if the generate object kind is anything other than *member*, or by the value of the member of the list if the generate object kind is *member*; in this case, the type of the list must be `[String]`, and the elements of the list must meet the lexical requirements for an identifier. If the generate object kind is *connection set*, the generate name is replaced with the name of any of the connectors in the connection set. Similarly, any name which is prefixed by the generate name and an underscore is replaced by a name prefixed by the name of the corresponding definition and an underscore. It is illegal if any name so altered is a type name.

8 Any name altered as specified in the preceeding paragraph must occur in a name definition within the generate definition. The type specification given by the name definition must consist of a definition designator that specifies the role of the altered name in the generated definitions. Each such altered name, after the alteration given, identifies an element of the design hierarchy that agrees with the definition designator given in the corresponding name definition containing the name before it is replaced by the altered name. In addition, if the generate object kind is `member`, the generate simple name must also appear in a similar name definition. If the altered name identifies a value (either by a property name or a variable name), the type of the value must be

the type given in the definition designator.

9 In the case of a table generate header, each time the generated definitions are replicated, all occurrences of any of the identifiers in the column name list in those definitions are replaced by the string corresponding to the identifier being replaced. See 2.4 [Tables], page 13, to determine the string corresponding to this identifier. Each occurrence of the generate simple name (if such a simple name occurs in the table generate header) is replaced by an integer literal that corresponds to the number of the row that is causing the replication. In addition, for each table generate header, two definitions are automatically generated. The first is a name definition, whose name is "<table name>_length" where <table name> is replaced by the name of the table, and whose type specification is `Int`. The second is a value defintion that sets the value of the name to be the number of rows in the table. It is an error if any name replaced, as given in this paragraph, produces another table generate definition.

10 All the replacements given above happen at the level of a lexical unit, as defined in the lexical rules of the language (see 9 [Lexical Rules], page 107, for more information). This means that occurrences of the column names in a string literal or within another identifier are not replaced.

11 It is an error if two different elaborations of the generate statements in a model result in the same named value being defined to be of two different types; i.e., the type of a named value is independent of the elaboration of the generate statements of a model. It is also an error is any name in a generate body is illegal within the normal rules for names (see 8 [Names], page 99, for more information about the legality of names in MHDL), regardless of whether the definition containing the name is duplicated. If the results of any model depend upon the order in which the generate staements are elaborated, that model is erroneous. See 3.4 [Elaboration], page 26 for more details on elaboration.

12 It is illegal if any expression (boolean or list expression) in the generate header refers to a value that is undefined when the generate definition is elaborated. See 3.4 [Elaboration], page 26, for the rules governing the elaboration of generate definitions. Expressions in the generate header are evaluated when the generate statement is elaborated.

2.4 Tables

1 A table gives the user the ability to group definitions together in a tabular form.

Example:

```
table parameters
```

```
  title "Parameter Values for Device";
  names   Parameter   | Value ;
       {{  gain       |  10'db'   ;
           noise      |  -5'db'   ; }}
end parameters;

for each row in table parameters generate
  Parameter = Value;
end generate;
```

Grammar Rules:

```
table_definition ::=
    table TABLE_simple_name
        [ table_header ]
        [ column_header ]
        column_name_list
        data_table
    end [ TABLE_simple_name ]

table_header ::=
    title STRING_expression ;

column_header ::=
    columns STRING_expression { '|' STRING_expression } ;

column_name_list ::=
    names identifier { '|' identifier } ;
```

2 A *table* is a format for giving information in tabular form. A table may have a title, which is a string (that may include carriage returns). A table is organized into rows and columns. The column header gives the text at the head of each column of the table. The column name list gives a name for each column. If the column header is not present, then the column header consists of the keyword *column*, and the identifiers found in the column name list enclosed in quotation marks, and separated by "|". If the column header is present, there must be as many string expressions as names in the column name list. The identifiers in the column name list must contain no duplicates.

3 The data table consists of a series of rows enclosed in double braces. Each row consists of a number of text strings, separated by the character "|" and terminated by the character ";". Either of these characters are represented within the text string by prefixing it with a backslash: "\". The backslash character is also prefixed by another

backslash to produce a backslash in the text. See 9.7 [Data Tables], page 115, for a formal definition of the data table.

4 To determine the text string corresponding to a given column name for a given row, the row is considered sequentially. The row defines a number of strings which are determined by eliminating the whitespace (spaces, tabs, and carriage returns) surrounding each delimiter ("|" and ";" characters not preceded by a backslash), and taking the remaining text between each pair of delimiters. If no such text exists, the string is empty. The string corresponding to a given column name is found by taking the strings defined by the row and matching them to the column names in the order in which they both occur. If there are fewer delimiters than column names, the corresponding string for the remaining column names is empty.

5 A table needs a corresponding generate definition to have an effect in the description. See 2.3 [Generate Definitions], page 9, for a discussion of the use of tables in generate definitions. If there is a name following the keyword `end` in a table definition, it must be the same name as that following the keyword `table`.

2.5 Duplicate Definitions

1 The definitions in a model may be from many sources, including packages and other models. A definition may be included in a model by more than one route; for example, if the model inherits two packages, both of which inherit a third. In such a case, the definition is only included once in the model that names the different packages and models from which it draws its definitions. If the same name maps to the same definition in two different visibility maps both of which are inherited by a given model or package, the mapping between the name and the definition is only included once.

2 If two structure definitions with the same structure name occur, then the structure definitions are combined according to the rules for combining such structure definitions. See 3.1 [Structure Definitions], page 17, for the rules for combining structure definitions with the same structure name. In all other cases, it is illegal if the same name is mapped to two different definitions. See 8.6 [Visibility], page 104, for the rules governing visibility and the definition of a visibility map.

2.6 Libraries

1 A library is a grouping of models and packages.

Example:

```
libraries user_mhdl, group_mhdl;
```

Grammar Rules:

```
library_list ::=
    libraries LIBRARY_simple_name
        { , LIBRARY_simple_name } ;
```

2 A *description unit* is either a package or a model. *Libraries* are collections of models and packages (description units) that are grouped together for the sake of name spaces and separation of a large project into smaller, independent pieces. Each description unit belongs to a library. The association of a description unit with a library is not specified in MHDL; rather, the implementation associates a description unit with a library by some processing step (such as syntactic analysis). For a given description unit, the collection of library names that appear in the library list of the description unit and the set of description units in the same library as the description unit must contain no duplicates.

3 Only one library list may appear in a description unit. Library lists are not transmitted between description units by inclusion lists as definitions are. Library names may be used to qualify description unit names (see 8.3 [Qualified Names], page 101). If no such qualification is found, the libraries in the library list are searched in the order that they appear. Two libraries always appear in the library list: the library containing the description unit that contains the library list(which is always identified by the name `Present`), and the library `MHDL_Standard`. If `Present` does not appear explicitly in the library list, it implicitly appears first in the list; similarly, if the library `MHDL_Standard` does not appear explicitly in the library list, it implicitly appears last in the list. If no library list is found in a description unit, an implicit library list consisting of `Present` and the library `MHDL_Standard` is assumed.

Chapter 3

Structure

The division of a complex design into interconnected pieces is recorded in MHDL in a *structure*. A structure is composed of uses of models called *components* that are related by *connectors*. A connector corresponds to some form of information flow between the components of a model. The form of the information flow is governed by signals and attributes attached to the connectors or to the structure that contains the components.

If a structure contains one or more components within it, then those components indicate other structures contained within another model. Collections of such related structures within structures that are used by other structures can be assembled into a *design hierarchy*. The parts of a design hierarchy may be indicated by names used in the different components, or in a separate definition called a *configuration*.

3.1 Structure Definitions

A structure records the division of a complex design into interconnected pieces.

Example:

```
structure Time_domain_behavior
  connectors Input, Output;
  attribute Value : Potential;
  Output.Value = D_band (Input.Value);
end Time_domain_behavior;
```

Grammar Rules:

```
structure_definition ::=
    structure STRUCTURE_simple_name
        [ inclusion_list ]
         [ structure_element_list ]
     end [ STRUCTURE_simple_name ]

 structure_element_list ::=
    structure_element ; { structure_element ; }

structure_element ::=
    structure_item
  | connector_clause
  | component_list

connector_clause ::=
    connectors [ connector_name_list ]

connector_name_list ::=
    CONNECTOR_simple_name { , CONNECTOR_simple_name }

structure_item ::=
    connection_list
  | abstract_property_definition
  | name_definition
  | value_definition
  | equation_definition
  | constraint_definition
  | association_definition
  | configuration_definition
  | generate_definition
  | table_definition

 component_list ::=
    components
        component_definition ; { component_definition ; }
    end components
```

2 A *structure definition* specifies the structure as a series of connectors that specify the interface to users of the structure, a number of definitions, a number of connection lists

that specify how the connectors of the components and of the enclosing structure are interconnected, and a number of components. The form of the structure is given by the components and the connection lists. The behavior of the structure is specified either using functions or by the components of the structure and how those components are connected. These components are given by a series of *component definitions*, each of which creates a part from the named structure (and any containing model) and specifies relationships between the connectors in the component and the structure which contains the component.

3 If two or more non-identical structures with the same name appear in a model, either directly or by inclusion, they are treated as a single structure with their definition lists, connector lists, and component lists textually combined. If two component definitions with the same component name appear in any component list of the same structure, the two definitions must have the same structure or model name; in this case, they are regarded as a single component definition with that component and structure or model name and with the definition lists textually combined; otherwise, it is an error. A structure may contain any definition that may appear in a model, with the exception of other structure definitions, class definitions, instance definitions, type definitions, dimension definitions, and unit definitions.

4 The properties that are defined in a structure definition are duplicated each time the structure is named in a component definition. This is done in the process of creating the parts in a design hierarchy (see 3.3 [Design Hierarchies], page 25. The properties are identified using the name of the structure (see 8.4 [Hierarchical Names], page 102).

5 If a structure name occurs following the keyword *end*, it must be the same as the structure name at the beginning of the structure definition.

3.1.1 Components

1 A component indicates a structure that will "replace" the component in the eventual design.

Example:

```
C5 :: Voltage_Limiter.Time_domain_behavior
definitions
    Low_th = -12.0 ;
    High_th = 12.0 ;
    Gain = 1.0 ;
    VMax_neg = -12.0;
    VMax_pos = 12.0;
```

```
end C5;
```

Grammar Rules:

```
component_definition ::=
    COMPONENT_simple_name ::
        [ STRUCTURE_OR_MODEL_OR_CONFIGURATION_name ]
        [ [ inclusion_list ]
        [ definitions definition_list ]
    end [ COMPONENT_simple_name ] ]

 definition_list ::=
    structure_item ; { structure_item ; }
```

2 A *component* is a use of a structure. A component creates a part of the definitions in the model that contains the structure indicated by the structure name (or by the configuration that names that component; see 3.3 [Design Hierarchies], page 25, for more details). The value of properties attached to names in these definitions may be set by definitions within the component, within the structure containing the component, or within the model containing the structure. See 5.2 [Property Value Definitions], page 57.

3 The structure, model, or configuration name (called the *structure indicator* in what follows) indicates a unique structure with which the component name is to be associated. The structure indicated by the structure indicator is determined as follows. The model named in the structure indicator is located; if it does not exist, the *indicated structure* is null. If a structure name is the simple name of the structure indicator, then that structure is located within the model; otherwise, the structure name of the structure containing the component is used. If this named structure does not exist, the indicated structure is null. If a configuration name is used, the name in the configuration body must be a structure name, and this structure is the indicated structure.

4 If the indicated structure is null, a configuration will be needed to specify the structure to be associated with the component. If a configuration is used to associate the component with a structure, the indicated structure is given by the configuration and not by the configuration, structure or model name in the component definition (see 3.2 [Configurations], page 22, for more detail). In either case, the component and the structure are treated as associated definitions; the properties of one may be read through the other (see 5.3 [Associations], page 58, for more information).

5 If two or more component definitions of the same name appear in the same structure, their definition lists are combined in the same manner as the definition lists in models

(see 2.5 [Duplicate Definitions], page 15, for more information).

6 If a component name appears after the keyword *end*, it must be the same name as that appearing at the beginning of the component definition.

3.1.2 Connectors

1 Connectors represent information that is shared by two or more components (either as an information flow or as a wave appearing at the connector).

2 *Connectors* are objects within a structure that represent information linkage between components within the structure and components that exist in a containing structure. Connection lists associate connectors defined in structures (called *used structures*) that are indicated structures in components that appear in a "parent" structure (called the *containing structure*); such connectors may also be associated with connectors defined in the containing structure itself.

3 Connectors have no type, as they have no value of their own; rather, the attributes and signals of the connectors have types and values.

3.1.3 Connection List

Grammar Rules:

```
connection_list ::=
    [ label ] connect
        connector_specification { , connector_specification }

connector_specification ::=
    CONNECTOR_OR_CONNECTION_LIST_name
```

1 The connection list associates connectors in the containing structure with connectors in a used structure and/or connectors in one or more used structures. This represents information linkage between components in the same structure, and between connectors in parents and children defined in a structure. The association has implications for property visibility and availability (see 5.3 [Associations], page 58). The values of the properties on the various connectors are set by the property assignments appearing in the structure, in the component or within the used structure, and are not necessarily dependent upon the connection list. In addition, the component name and the name of the structure indicated by the structure name are associated.

2 A connection list consists of a number of connectors and/or connection list names, where the name of a connection list is a label that appears before the desired connection list. Each connector and each connection list label has an associated connection set. The connection set of a given connector, or of a given connection list label, is the set of connectors from throughout the design hierarchy to which a particular connector is connected. More precisely:

1. The connection set of each connector includes itself.
2. The connection set of each connection list label is the same as the connection set of any of the connectors or connection list labels in the connection (which should all be equal).
3. If a connector or a connection list label appears in a connection list, then the associated connection set of each connector or connection list label in the connection list is the union of the connection sets of all the connectors and connection list labels in the connection. This is applied recursively, until the connection sets reach their maximal size.
4. No other connectors are in the connection set of any connector.

3 **Note:**

> Because recursive structure definitions are prohibited, this algorithm always terminates. See 3.3 [Design Hierarchies], page 25, for more details.

4 Each connector is associated with the connectors in its connection set. For the purposes of analysis, the connectors in a connection set are all deemed to be connected at a single point, or terminal; this terminal may be identified by the name of any connection list that is associated with that connection set.

3.2 Configurations

1 A configuration optionally identifies the structure to be used for a given component.

Example:

```
configuration use_kcl
  for electrical_breakdown
    for op_amp use kcl_op_amp;
```

```
      for mixer use kcl_mixer;
      for transmitter
        if accuracy == high then use transmitter
        else use simple_trans
      end transmitter;
    end electrical_breakdown;
  end use_kcl;
```

Grammar Rules:

```
   configuration_definition ::=
      configuration CONFIGURATION_simple_name
          configuration_body
      end [ CONFIGURATION_simple_name ]

  configuration_body ::=
      configuration_clause ; { configuration_clause ; }

  configuration_clause ::=
      simple_configuration
    | complex_configuration

  simple_configuration ::=
      for COMPONENT_OR_STRUCTURE_name
         use MODEL_STRUCTURE_OR_CONFIGURATION_name

  complex_configuration ::=
      for COMPONENT_OR_STRUCTURE_name
          configuration_alternative
      end [ COMPONENT_OR_STRUCTURE_name ]

  configuration_alternative ::=
      absolute_configuration
    | if_configuration
    | case_configuration

  absolute_configuration ::=
      [ use MODEL_OR_STRUCTURE_OR_CONFIGURATION_name ]
      { configuration_clause ; }

  if_configuration ::=
```

```
        if BOOLEAN_expression then
           absolute_configuration
        { else if BOOLEAN_expression then
           absolute_configuration }
        else absolute_configuration

    case_configuration ::=
        case expression of '{'
            component_alternative ; { component_alternative ; }
        else
            absolute_configuration ; '}'

    component_alternative ::=
        pattern -> absolute_configuration
```

2 A *configuration* is a mechanism for precisely identifying the structure that is associated with a component. A series of absolute configurations for each component may be given. If more than one absolute configuration is provided, these absolute configurations are given either in the form of an if configuration or a case configuration. The absolute configuration that indicates the absolute configuration to be used is chosen in the same manner as the expression for an if expression or a case expression, respectively. See 7.10 [Simple Expressions], page 91 for more details on these two expression forms.

3 A list of configuration clauses must either contain all component names or all structure names after the keyword *for*. A configuration clause containing a structure name is called a *structure configuration*; a configuration clause containing a component name is called a *component configuration*. A structure configuration identifies a structure that contains components to be bound to structures; a component configuration binds components to structures. If the configuration body contains configuration clauses with component names, then a unique structure (called the *configured structure*) is identified in one of two ways:

- If the configuration definition occurs in the definition list of a structure, then the structure containing the configuration definition is used.
- If the configuration definition is used by having the configuration name appear in a configuration clause of another configuration, then the structure name that appears in the configuration clause naming the configuration definition is used. In other words, a configuration in a structure configures the structure in which it appears.

4 In other cases, the configured structure is identified by the structure name in the

configuration clause that contains the configuration clause list. Such a structure name can be either a simple name or a hierarchical name. It is an error if a component name that appears in a configuration clause does not have a corresponding component definition (i.e., a component definition with the same name) in the configured structure.

5 If a structure configuration contains one or more component configurations, then the component configurations map the components in the structure indicated by the structure name in the configuration clause to other structures.

6 The appearance of a component name in a configuration being used to control the elaboration of a design hierarchy always overrides the structure indicator in a component definition.

7 The appearance of a configuration name in a simple configuration or an absolute configuration signals that the named configuration will define the bindings of components to structures in the children of that component. Such a configuration name must be a simple name, and the configuration must therefore be visible to the component being configured.

8 The configuration affects the elaboration of the design hierarchy, and is evaluated during the elaboration of the design hierarchy that it describes (see 3.4 [Elaboration], page 26, for more details). It is illegal if the creation of the design hierarchy requires that an expression occurring in a configuration be evaluated and that expression contains a reference to something that has not been elaborated when the expression is evaluated.

9 It is an error if a structure configuration immediately contains another structure configuration. If a component configuration immediately contains one or more component configuration, then the components bound by those component configurations are those within the structure named by the outer component configuration. Otherwise, all component configurations bind components in the structure identified by the immediately enclosing structure configuration, or by the configuration definition itself, as defined above.

10 It is an error if multiple configurations bind the same component and / or structure, as determined by the above rules. Since structural recursion is forbidden in MHDL, recursive configurations are likewise forbidden.

11 The name after the keyword *end* of complex configuration must be the same as the structure or component being configured by that clause.

3.3 Design Hierarchies

1 A model contains a number of definitions (see 2 [Models], page 5, for a discussion of models). A *part* of a model is a collection of entities and properties created from the

definitions in a model (see 3.4 [Elaboration], page 26, for more details). Each component creates a part from the definitions in the model that contains the structure indicated by the structure indicator or indicated by the configuration for that component. Thus, each structure may create a number of parts, each of which may contain structures that create more parts.

2 A *design hierarchy* is a representation of an entire system that consists of a number of parts. Each part consists of a number of properties (parts are also said to contain definitions; in actuality, if a part contains a definition this really means that the definition is contained in the model from which the part was created). A design hierarchy begins with a single part of a single model, which is called the *root* of the design hierarchy. Each part created by a component in a structure in the root is included in the design hierarchy; such parts are called the *children* of the root, which is the *parent* of those parts. This process continues with the structures in the children. More formally:

- The root is in the design hierarchy.
- If a part is in the design hierarchy, then a part is created for each component in each structure defined in the existing part. These parts are in the design hierarchy.
- No other part is in the design hierarchy.

3 A design hierarchy therefore consists of a number of parts. In many cases, the names in the different parts will be associated (see 5.3 [Associations], page 58, for a more general discussion). Information is maintained within the design hierarchy by property values. See 5.2 [Property Value Definitions], page 57, for more discussion about property values and their definition.

4 It is an error if any of the children of a part are created from the same model as the parent part. This restriction covers the entire design hierarchy, i.e. the transitive closure of the children relationship. Thus, recursive structures are prohibited.

3.4 Elaboration

1 Elaboration is the process of creating a design hierarchy, consisting of a number of parts, from a specific model. This process consists of identifying the values that will be defined in the various parts, and reflecting the effect of any generate definitions in the various models.

2 In MHDL, many names may have properties attached to them. Thus, names in different parts may have properties with different values attached to them. The fact that different values occur in the different parts of a design hierarchy created from the same

model requires that a placeholder be created for each name to which a property is attached. Such a placeholder is called an *entity*. An entity is any name to which a property is attached.

3 The process of identifying names to which properties are attached, and making these property values unique, is called *creating the entity* and *creating the property*, respectively. This may or may not require the reservation of memory in a computer implementation, depending upon that implementation. The definition of the process of creating entities and properties is the recognition that the values attached to those entities have a unique existence with respect to the part that contains them, and that the values may be different in different parts created from the same model.

4 **Note:**

> In general, the use of nouns for various parts of the MHDL language refers to the entity that is created from a name during elaboration. For example, the term "structure" is used to denote the entity created from the name in a structure definition. In some cases, this correspondence does not hold; the entity created from a model name is a part.

5 Part of the creation of an entity or property (called the *created property*) is the creation of the associations between the created property and any entity or property with which it is associated (called the *associated property*; see 5.3 [Associations], page 58, for more information). If the associated property has been created, the associations are elaborated by making properties attached to the created and associated names visible in their respective parts. If the associated property has not yet been created, then creation of the associations is deferred until it is created.

6 Elaboration begins with a specific model, by creating a part for that model. Creating a part for the model effectively creates the entire design hierarchy. This part is the root of the design hierarchy.

7 A part is created as follows:

1. The part is created, and given a unique existence in the design hierarchy. Parent relationships are established.

2. The definitions of the model are used to create the entities and properties of the part.

3. Any generate definitions are elaborated, and the definitions that are replicated during the elaboration of the generate definitions are used to create the entities and properties for those definitions, as follows:

- The expressions in the generate header are evaluated. It is an error if these expressions require the value of any named value or entity that has not been created. This evaluation results in the creation of an iteration set, as specified by the semantics of the generate definition. See 2.3 [Generate Definitions], page 9, for the definition of these semantics.
- For each element in the iteration set, the definitions and properties within the generate definition are duplicated, their entities and properties are created, and the entities and properties are added to the part. Any generate definition within these definitions is elaborated, recursively.

4. A new part is created for each component definition in the part being elaborated, using the model indicated by the structure indicator in the component definition. Child relationships are established. This process proceeds recursively until termination; it is an error if it does not terminate.

3.5 Layouts

1 A layout is represented as the value of the attribute `layout`, which may be attached to any structure. The attribute `layout` is of type `Topology`, which is given in package Standard (see 10.3.5 [Topology], page 162, for this definition). This definition reflects the PDES Topological standard ([1]). An implementation is required to conform to the PDES standard when implementing the MHDL data type `Topology`.

Chapter 4

Types and Classes

A *value* in MHDL is an identifiable mathematical quantity or named substitute for such a quantity. A *type* is a set of values, for example a range of real numbers or integers. MHDL includes the normal types of most programming languages, including integers, floating point numbers, characters, and strings. The exact definition of these types are found below. In addition, the following types are included in MHDL:

- Functions. Functions are normal values in MHDL, and functions with the same type specification belong to the same type.
- Physical quantities. Physical quantities, such as time, length, area, energy, and other physical quantities are included in MHDL.
- Signals. Function types with a domain of a subset of the physical types (including time and frequency) and a range of any physical type are given a special status in MHDL, and are called *signal types*.

Note:

> In general, predefined types in MHDL are not materially different from user defined types. This gives the language considerable flexibility, including the ability for the user to add his types. The primitive function facility is vital to this mechanism. See 7.10.9 [Primitive Values], page 96, for more details.
>
> Because of this feature, all of the types below (including predefined types) are also defined in package Standard. See 10 [Standard Definitions], page 119, for the exact definition of package Standard.

4.1 Type Definitions

1 A type is a set of values. In most cases in MHDL, we are interested in named sets of values. Such sets are specified by a *type definition*. A type definition specifies a name for a type, and (possibly in a separate definition) a set of values that actually defines the type itself.

2 Each type is represented either by its header or by a type specifier. A *type header* consists of the name of the type and a series of polymorphic names that serve as placeholders for the types out of which the named type is built. A *type specifier* gives the complete definition of the type, including the mechanisms for constructing the type out of its components. A *type specification* is a type specifier that may also be used in type headers to designate the building blocks out of which a complex type is built. In MHDL, type agreement is by structure. Two types that have the identical type specifier, but different type names, are the same type.

3 MHDL supports object oriented development with the concept of *type classes*. A type class is a collection of name and value definitions (see 6.1 [Name Definitions], page 62). Each type belonging to the class must implement all the functions in the class definition. Such implementations are given in *instances*. A type must have one instance for each class to which it belongs.

4 Like functions, types can be polymorphic. For example, a list may be of any type — lists of integers and lists of physicals are still both lists. Constructs used for the definition of polymorphic types are given below.

5 Expressions may occur in various places in type, unit, dimension, class, and instance definitions. It is an error if these expressions contain elements which are dependent upon values which may vary from part to part; see 3.3 [Design Hierarchies], page 25, for a definition of parts. In effect, types, units, dimensions, classes, and instances must be independent of part distinctions.

6 **Note:**

> The MHDL type system is based upon the Hindley-Milner type system which is similar to that used in the programming language Haskell. The description in this document is the definition of the MHDL type system; however, implementation and other information may be found in the standard papers on the subject ([3, 6, 5, 4, 15]).

4.1.1 Specified Types

1 A specified type definition gives the type header and the set of values for a given type.

Example:

```
type VHDL_Signal (a) = [ ( Time_dim (Integer), a ) ] ;
```

Grammar Rules:

```
specified_type_definition ::=
    type type_header = type_specifier [ derivation_clause ]
        [ where context ]

type_header ::=
    TYPE_simple_name [ polymorphic_name_list ]

polymorphic_name_list ::=
    ( POLYMORPHIC_simple_name { , POLYMORPHIC_simple_name } )

type_specifier ::=
    type_specification
  | constructed_type

type_specification ::=
    type_component [ -> type_specification ]

type_component ::=
     CONSTRUCTOR_simple_name ( type_component { , type_component }
)
  | definition_designator
  | TYPE_name
  | POLYMORPHIC_simple_name
  | ( )
  | ( type_specification )
  | ( type_list )
  | '[' type_specification ']'
  | '[' ']'
  | '[' ']' ( type_specification )
  | ( -> )
  | ( -> ) ( type_list )
  | ( , { , } )
  | ( , { , } ) ( type_list )

definition_designator ::=
```

```
    model
  | structure
  | constraint
  | equation
  | function
  | connector
  | component
   | property ( type_specification )
  | variable ( type_specification )
  | label
   | definition
  | type

type_list ::=
    type_specification , type_specification
        { , type_specification }

constructed_type ::=
    constructor_list [ with '{' constructor_definitions '}' ]

constructor_list ::=
    type_constructor { '|' type_constructor }

type_constructor ::=
     CONSTRUCTOR_simple_name [ ( field_list ) ]
  | type_component CONSTRUCTOR_operator type_component

field_list ::=
    [ FIELD_simple_name :: ] type_component
        { , [ FIELD_simple_name :: ] type_component }

constructor_definitions ::=
    value_definition ; { value_definition ; }

derivation_clause ::=
    deriving ( class_list )

class_list ::=
    CLASS_name { , CLASS_name }

context ::=
    class_constraint
```

```
    | '{' class_constraint { , class_constraint } '}'

class_constraint ::=
    instance_spec
```

2 A specified type definition establishes an identifier as a type and the set of values that the type represents. The set of values is given by a type specification, which identifies the set of values as described below.

3 If a derivation clause exists in the type definition, then *derived instance definitions* are automatically generated for the classes in the derivation clause. If a class name appears in the derivation clause, then all of its superclasses are treated as appearing in the derivation clause. The only names that may appear in a derivation clause are `Eq`, `Ord`, `Ix`, `Enum`, `Text`, and `Binary`. These class instances are generated in an implementation dependent manner, in accordance with the definition of the functions given in package Standard; where Ord is defined by the normal lexicographic ordering. Any component types of the type being defined must have an instance defined for each class in the derivation clause.

4 The occurrence of a polymorphic name in a type header represents the definition of a *polymorphic type*. A polymorphic type is a type which is made up of other types, where the other types are not yet specified. Whenever the type name of the polymorphic type occurs in a type specification, it must be followed by the same number of type components as there were polymorphic names in the type header with that same name (except within an instance header); this is called a *specialization* of the polymorphic type. In the case of an instance header, all instances of the same class must require the same number of type components to be added to the type component in the instance header to create a complete specialization of the polymorphic type. Any such polymorphic type occuring in an instance header that requires more than zero additional type components must be a constructed type.

5 An occurrence of a specialization of the polymorphic type is defined to be a textual replacement of the specialization with the type specification in the type definition of the polymorphic type (surrounded with parentheses to avoid ambiguity), with each polymorphic name in the type specification of the polymorphic type replaced with the corresponding type component in the specialization (again surrounded by parentheses to avoid ambiguity). Within the name definitions of a class definition, a type listed in the class type list may be specialized by the addition of type components; in this case, within each instance of the class, it is as if these type components are added to the list of type components of the corresponding type component in the instance header. This completed type component is then used to specialize the type specification in the instance header; it is an error if this completed specialization does not have the same number of type components as polymorphic names in the type header with that same

name.

6 The occurrence of a context following the type specifier places constraints on the type specifications that may be substituted for a polymorphic name during the substitution of a specialization with its corresponding type specification. If an instance spec occurs in a context, then an instance which is a specialization of that instance spec must exist for the class named in the instance spec; it is an error if the type name in the type header does not occur in the instance spec in the context. See 4.1.4 [Class Instances], page 39, for the definition of instance spec and an explanation of instance definitions. The polymorphic names in a type header must be distinct, and each polymorphic name in the context must be used in the type or class definition in which the context occurs.

7 A *constructor* is a syntactic element that has a name which specifies a number of values that are made unique by the use of the name. If a type specification takes the form of one or more constructors, then each constructor in the type specification specifies a set of values, and the type is the union of those sets. Each constructor has a name or operator and a number (possibly zero) of type specifications. The set specified by each constructor is the Cartesian product of the sets specified by the type specifications, prefixed by the constructor name or separated by the operator (which eliminates ambiguity between two structurally identical types which are conceptually different).

8 A constructor name is associated with a value, as is the constructor operator. The value has type:

```
a1 -> ... an -> T where con
```

9 where `con` is the context associated with the type definition, `T` is the type header (including the polymorphic name list, if any), and one construct of the form "`ax ->`" is added to the type signature for each type component in the type constructor, in the same order as they occur in the type constructor.

10 The value associated with the constructor name is a function with the type given above. In the absence of a constructor definition which gives a value to the constructor name, this value is a function that takes values of the types given in the type specification above and produces a value of the constructed type T whose values are given by the function application arguments.

11 If constructor definitions appear in the constructed type, there is at most one value given to each constructor name; it is an error if a value is given to any other name than one of the constructor names. The value definition for a constructor name must be of type:

```
a1 -> ... an -> Bool where con
```

12 where the names are given as above. Where such a definition appears, the value of the constructor name is:

```
case exp1 of {
  True -> exp0 } ;
```

13 Where exp1 is the appropriate application of the value definition for the constructor name, and exp0 is the value defined above for a constructor name with no corresponding value definition.

14 If a type specification is of the form "a -> b", or "(->) (a,b)", where a is a type component and b is a type specification, then the specified type consists of the set of functions defined in MHDL that take an argument of type a and return a result of type b. See 4.2.3 [Function Types], page 41, for more details. This is the only form of type specification in which a definition designation may occur. Indeed, the only place where a definition designation may occur is in the name definition of a primitive value (see 7.10.9 [Primitive Values], page 96, for more details).

15 A definition designator may only occur in a type specification in a name definition. Name definitions inside a generate definition are a special case; (see 2.3 [Generate Definitions], page 9, for more information; the variable definition designator may only occur within a generate definition. Any value definition associated with any other name definition must have an expression which is a primitive value specification. If the definition designator is *property*, then any property name supplied as a value for that definition designator must be of the type given in the following type specification. See 7.10.9 [Primitive Values], page 96, for more information.

16 If a type component is of the form "(a1, a2, . . . , an)" where there are at least two type components present, or "(, , ... ,) (a1, a2, ..., an)", then this is a tuple type. The set of values is the Cartesian product of the type components, with each element prefixed with the name "tuple[n]", where "[n]" is replaced by the number of elements in the tuple (i.e., n in "an" above).

17 If a type component is of the form "(a)", where a is a type, then this is a parenthesized type, and the set of values is the same as those identified by a.

18 If a type component is of the form "[a]", or "[] (a)", the set of values is that given by the type "[]"; see 4.2.4 [Lists], page 41, for more details.

19 If a type component is a type name, then the set of values corresponds to that defined by the type name in the type specifier.

20 If a type specification is a polymorphic name, then the set of values consists of the union of all types that may be substituted for the polymorphic name during the substitution of a type specification in a specialization, according to the rules above.

21 If the type specification consists of empty parentheses, then it is the null type, and consists of the single value written "()".

4.1.2 Primitive Types

1 A primitive type specifies a type which is in some sense basic, or predefined. Primitive types may only be defined in package `Standard`.

Example:

```
type Int;
```

Grammar Rules:

```
type_definition ::=
     primitive_type_definition
   | specified_type_definition

 primitive_type_definition ::=
    type type_header
```

2 A primitive type definition identifies a name for a type. Primitive type definitions may only occur in package `Standard`. The set of values for the types defined by primitive types are given in the various definitions in the LRM.

4.1.2.1 Record Types

1 Any type consisting of one or more constructors whose field list includes field names is called a record type. Each field name is associated with one of the type components in the field list. In a value of record type, the field name identifies that portion of the value which is of the type given by the type component directly following the field name, and which is associated with that specific position in the field list in which the field name occurs. This portion of the entire value may thus be extracted using the field name. See 8.2 [Record Names], page 100, for a discussion of the record name and its use in extracting this value.

2 If a field name occurs multiple times in a given constructed type definition, the type component that follows the field name must be the same in all cases. A field name may

occur multiple times in a constructed type definition, and the same field name may occur in different constructed type definitions.

3 **Note:**

> There are two ways to gain access to the fields of any record: first, by means of a pattern match (see 6.3 [Patterns], page 72, for more information on pattern matching); second, by means of the field name. Because the record type is part of the general notion of constructors in MHDL, each field does not require a name if the field name is not going to be used to extract the field value.

4.1.2.2 Enumeration Types

1 If a type specification consists of one or more constructors consisting solely of constructor names, then this is an *enumeration type*. Each enumeration type forms a total order which is defined by the order of the constructor names in the type definition. Each enumeration type is automatically an instance of class `Enum` (see 10.1.7 [Enum], page 122), and the functions in that class are implemented according to the total order defined by the enumeration type definition, where the ordinal number of the first constructor is zero. Each enumeration type is also automatically an instance of the class `Ix`, and may thus be used as an array index.

4.1.3 Type Classes

1 A type class is a list of functions which must be implemented for each instance of the class.

Example:

```
class Eq (a) {
  (==)  :: a -> a -> GenBool;
  (/=)  :: a -> a -> GenBool;
  x /= y  = not x == y ; } ;
```

Grammar Rules:

```
class_definition ::=
    class_header [ class_body ] [ where context ]
```

```
class_header ::=
    class CLASS_simple_name class_type_list

class_type_list ::=
    ( type_specification { , type_specification } )

class_body ::=
    '{' class_definition_list  '}'

class_definition_list ::=
    function_definition ; { function_definition ; }

function_definition ::=
    name_definition
  | value_definition
```

2 A type class is a group of related name definitions, possibly accompanied by some value definitions. This group is given by a class definition. Each name in a name definition is said to name an *operation*, or *method* of that class. Each instance of the class may define an implementation of any or all of the name definitions. The value definitions in a class definition constitute default implementations of the functions named in the name definitions. An instance of the class may redefine these functions if desired. All value definitions in a class definition must have explicit name definitions in the same class definition.

3 A function name that occurs in several different value definitions in different instances must occur in a name definition in the class definition named in the instance spec of all the instance definitions. Each of the value definitions for that function name must be either in a class definition or a class instance, and may occur only once in that class definition or class instance. Such a function name is said to be *overloaded*.

4 The context of the class establishes relationships between the class being defined and other existing classes. Each class named in the context is said to be a *superclass* of the class in which the context occurs; this class is a *subclass* of its superclasses. The superclass relationship is transitive; superclasses of the superclasses of a class are also superclasses of the class. Each instance of the class may use instances of its superclasses in the value definitions within the instance definition; i.e, for each instance spec in the context of the class, and for each instance of that class, an instance must exist whose instance spec is the same as the instance spec found in the context after the substitutions for the instance are made in the class header and context.

5 Polymorphic names in the class header may be used as the names of types that take

parameters in the name definitions and value definitions of the class. This use of polymorphic names is unique to class and instance definitions. The number of arguments of a polymorphic name so used as a type name must be the same in all name and value definitions in the class definition, and all the instances of the class. Each polymorphic name in the class header must occur in all of the name definitions in the class.

6 **Note:**

> The above interpretation of a class constraint is identical with interpretations of other class constraints on different polymorphic names. The paragraph above defines the meaning of the term superclass. See 4.1.4 [Class Instances], page 39 for a complete definition of the meaning of a context in a type class and instance.

4.1.4 Class Instances

1 A class instance gives values for the names defined in a class definition.

Example:

```
instance Eq (Int) {
  (==) = inteq; };
```

Grammar Rules:

```
instance_definition ::=
    instance_header
        [ '{' instance_body '}' ]
        [ where context ]

instance_header ::=
    instance instance_spec

instance_spec ::=
    CLASS_name ( type_specification { , type_specification } )

instance_body ::=
    value_definition ; { value_definition ; }
```

2 A class instance is a group of value definitions that corresponds to a type class. Each

name that is given a value in any of the value definitions in the instance body must be defined in a name definition in the type class whose name occurs in the instance spec (called the class of the instance, or the instance's class).

3 It is illegal for two instances to have the same instance spec, or for the sequence of type specifications in the instance spec to resolve to the same sequence of types.

4 There must be a type specification in the instance spec corresponding to each type specification in the class header of the instance's class, where corresponding means occurring in the same textual order. The type specifications in the instance spec must be a specialization of the respective type specifications in the class header.

5 A name definition is formed for each value definition in the instance body as follows. The function name in the name definition is the same as the function name in the value definition in the instance (called the instance value). The type component of the name definition is the specialization of the type component in the name definition in the instance's class with the same function name, which is formed by replacing each polymorphic name in the class header of the instance's class with the corresponding type specification in the instance spec. When this specialization is formed, all the class constraints in the instance spec must be met as well as the class constraints in the class header of the instance's class. The value definition is the instance value.

6 If a function name in one of the name definitions in the instance's class definition does not occur in the instance, then the value definition in the instance's class is used. It is not an error if a function name in one of the name definitions in the instance's class does not have a value definition with the same name in the instance.

7 **Note:**

> A consequence of the above rules is that the name definition for any value definition in an instance definition is a specialization of the type specification of the corresponding name definition in the class definition.

4.2 Predefined Types

1 A number of types in MHDL are predefined. In MHDL, there is no semantic distinction between predefined and user defined types. Therefore, all predefined types have definitions in MHDL, and are contained in package Standard. See 10 [Standard Definitions], page 119, for more details and a copy of package Standard in MHDL. A summary of the predefined types and their characteristics appears below. In some cases syntactical shorthand is used for the constructors to facilitate readability. This syntactic shorthand is also given below.

4.2.1 Booleans

1 Booleans represent the traditional binary values of `True` and `False`. The type `Bool` is an enumeration type consisting of those values. See 10.2.1 [Bool], page 138, for more information on the operations defined on booleans.

4.2.2 Characters and Strings

1 The type `Char` is an enumeration type. The members of this enumeration type are given by the standard ISO 8859-1. See 9.1 [Character Set], page 107, for a summary of this standard.

2 A string is a list of characters. The syntactic convenience of representing this list as a number of printable characters in double quotes, i.e. `''A string''`, is allowed. This is equivalent to the list consisting of the characters between the double quotes, in order: i.e., ['A', ' ', 's', 't', 'r', 'i', 'n', 'g'].

4.2.3 Function Types

1 A function is a mechanism for producing a value given another value. The most common function is that defined by MHDL syntax in the form of a name definition (see 6.2.1.1 [Functions], page 67). A function may thus be thought of as a set of mappings from one value to another. A function type is a set of functions, or a set of sets of mappings. Function types are defined by the type class `Function` in package Standard.

2 The apply function has special syntax in MHDL, which consists of the function name, followed by the arguments to the function enclosed in parentheses and separated by commas: f(a) represents the function f applied to a.

4.2.4 Lists

1 Lists are represented in package Standard by the type "[]". The constructor `Nil` is represented by the syntax [], and denotes the empty list. The colon - minus character `:-` represents the normal cons function of other list programming functions. There are a large number of standard functions on lists: see 10.3.2 [List], page 149, for more details.

2 A list may be represented by a list of values separated by commas: the list `[1, 2, 3]` is the same as `(1 :- (2 :- (3 :- [])))`.

4.2.5 Tuples

1 Tuples are arbitrary groupings of data values. For each n, where n is greater than 1, there is an implicit type definition in package Standard as follows:

```
type tuple<n> = tuple<n> (a1, ..., an)
```

2 where the notation `<n>` represents textual replacement of the number n. The syntax for this is to enclose the values in parentheses: (1, "a string", "k") is a tuple consisting of an integer, a string, and a character.

4.2.6 Arrays

1 Arrays can be thought of as specific representations of functions. An array is a function that takes a value of an index type (a type which is a member of class `Ix`) and returns the value of the array at that position. Arrays are constructed by giving a number of associations, or values for each element of the array in a given location; one data element must be given for each position within the bounds of the array. The element type of an array may be any type, including an array. See 10.3.3 [Array], page 155, for a complete definition of the functions available for building and modifying arrays.

2 All enumeration types are members of class `Ix`, where the range function returns a list of the enumeration values, in order, and the index function returns the position of the enumeration value in the original definition of the enumeration type.

3 A monad type `StArray` is provided for the user to specify functions that use arrays in a fashion that specifically allows for incremental update; see 7.10.7 [Generators], page 94, for a more detailed discussion of monads. Implementors are encouraged to provide sequential, non-copying access to arrays when used with this type.

4.2.7 Null Type

1 The null type has a single value, represented by an empty pair of parentheses with no whitespace: `()`.

4.2.8 Integers

1 Integers are implemented in MHDL as the type `Int`, whose elements are implementation defined. Any MHDL implementation must at least provide elements in the range -2147483647 to 2147483647, inclusive. An implementation can provide other fixed integer types.

4.2.9 Rationals

1 The rational numbers are provided by the type `Ratio`. Rational numbers are represented as a tuple of two integers, representing the numerator and the denominator. All rational values are provided in their lowest form, with no divisor common between the numerator and denominator except 1. See 10.1.14 [Ratio], page 126 for a definition of the operators defined on rational numbers.

4.2.10 Reals

1 The real numbers are represented by the type `Float`. `Float` is constrained to cover at least the numeric values of the IEEE floating point standard 754 ([9]) for double precision numbers.

2 Floating point numbers in MHDL are also physical numbers, of type `Dimensionless` (which is the same as type `Float`). All arithmetic is defined on such physical quantities for floating point numbers; therefore, no numeric operations are defined on type `Float`. Instead, these operations are defined on type `Physical`, to which floating point numbers are coerced as necessary.

4.2.11 Complex Numbers

1 A complex number is a tuple of two real numbers. The Cartesian coordinate system is represented by a tuple of two numbers in the class `Real`, and the polar coordinate system is represented by the type `Polar`. See 10.1.16 [Complex], page 128, for a complete definition of the operations on complex numbers.

4.2.12 Physical Types

1 A *physical quantity* is a numeric quantity and an associated dimension that refers to the physical sense of the numeric quantity (for example, length or mass). A physical quantity may be represented by a physical value in which the numeric quantity is an arbitrary expression and the dimension is expressed in units. A unit refers to a unit of measure (such as meter or gram). A *physical type* is a set of values that measure a physical property.

2 The MHDL physical types comprise the type `Physical` and the dimension types. Physical quantities of arbitrary dimensionality are represented by the predefined type `Physical` that is defined in package Standard. Physical quantities of a particular dimension are represented by a dimension type.

3 The two kinds of physical types are closely related. Both are implementation defined types that are polymorphic with respect to the numeric quantity. Predefined operations on values representing physical quantities are defined for type `Physical`; implicit coercions allow operations involving values of dimension types. See 4.2.12.1 [Dimension Definition], page 44 for details on dimension types and their relationship to type `Physical`.

4 For real numbers, arithmetic and mathematical operations are defined on the physical form of the number rather than the non-physical form. Specifically, there is no instance for type `Float` in class `Real`; these operations are provided by type `Physical`. See 10 [Standard Definitions], page 119, for more details and the definitions in question.

5 Addition and subtraction of two physical values require that the dimensions be the same. Multiplication, division, and exponentiation perform the appropriate alteration of the units in the resulting value; for example, any exponent must be of dimension `Dimensionless`. Some other operations have restrictions on the dimensions that can be associated with the physical value applied to them. Other operations leave the units in the physical value untouched or return a result that is a pure number with no units.

6 A value of type `Physical` may be constructed by specifying a physical value, or by (implicitly or explicitly) applying the coercion function `toPhysical` (defined in package Standard) to an expression. The former method constructs a quantity of a particular dimension; the latter method constructs a dimensionless quantity. See 7 [Expressions], page 79, for the syntax of a physical value.

4.2.12.1 Dimension Definition

1 A dimension definition defines a physical dimension in MHDL.

Example:

```
dimension length;
dimension area = 'length ^ 2';
```

Grammar Rules:

```
dimension_definition ::=
    dimension DIMENSION_identifier
        [ = DIMENSION_designation ]
        [ DIMENSION_variations_clause ]

designation ::=
```

```
    ' designation_indicators '

designation_indicators ::=
    NUMERATOR_indicators [ '/' DENOMINATOR_indicators ]
  | ( designation_indicators )

indicators ::=
    indicator { '*' indicator }
  | ( indicators )

indicator ::=
    DIMENSION_OR_UNIT_name [ ^ indicator_exponent ]
  | ( indicator )

indicator_exponent ::=
    [-] integer_literal [ '//' integer_literal ]
  | ( indicator_exponent )
```

2 A *dimension* is an abstract physical property, such as length or power. The dimension introduced by a dimension definition is either a *basic dimension* (in which case no dimension designation occurs in the dimension definition) or a *derived dimension* (in which case the dimension is defined in terms of other dimensions). The definition of a basic dimension is restricted to appearing in package Standard, hence an implementation can define additional basic dimensions, but the user cannot.

3 A *designation* is an algebraic description of a dimension; it is expressed as a fraction comprised of *indicators* corresponding to the constituent dimensions , their multipliers, and their exponent values; exponents apply to both the name and the multiplier, if present. A *dimension designation* is an abstract description of a dimension that is used to define a dimension (for example, `length/time`). A *unit designation* is a specific realization of a dimension that is used to define a derived unit or to construct a physical value (for example, `meters/second`). The names in a dimension designation must denote dimensions; similarly, the names in a unit designation must denote units. It is an error for a designation to contain both dimension names and unit names. It is also an error if a definition designation contains multipliers.

4 The names in a designation are distinct from those names used elsewhere in a description; thus, a unit may have the same name as a property (or other MHDL language feature) without this being an error. Indeed, for dimensions a type with the same name as the dimension is implicitly defined, as discussed below. Thus, the unit "name space" is separate from the ordinary MHDL "name space".

5 The expression following the ^ symbol in a dimension indicator is the exponent of the

dimension to the left of the symbol. The expression representing an exponent must be a static integral or rational value of one of the following forms: an integer literal, or a rational value constructed from two integer literals using the `Ratio` constructor `//`. See 9.5 [Literals], page 110, for the syntax of an integer literal; see 10.1.14 [Ratio], page 126, for information on type `Ratio` and its constructor.

6 The dimension `Dimensionless` is treated specially. A dimension of `Dimensionless` indicates that there is no physical property associated with the numeric quantity. The dimension `Dimensionless` may not appear in a dimension designation. `Dimensionless` is equivalent to `Float`, as it is defined in package Standard. It is an exception to the creation of coercion functions as defined below; no coercion functions are created for it. In addition, the following implicit type definitions are not introduced for dimension `Dimensionless`. The component type serves as the implicit type for `Dimensionless`; for example, instead of `Dimensionless(Float)`, `Float` is used.

7 A dimension definition introduces an implicit type of the same name, as well as three implicit functions; these types are called *dimension types*. Their definitions are of the general form:

```
type DIMENSION_identifier_dim (a) = DIMENSION_identifier_dim (a);
type DIMENSION_identifier = DIMENSION_identifier_dim (Float);

coercion DIMENSION_identifiertoPhys ::
      DIMENSION_identifier_dim (a) -> Phys (a);
DIMENSION_identifiertoPhys =
      primitive "DIMENSION_identifiertoPhys";

coercion PhystoDIMENSION_identifier ::
      Phys (a) -> DIMENSION_identifier_dim (a);
PhystoDIMENSION_identifier =
      primitive "PhystoDIMENSION_identifier";

IsDIMENSION_identifier :: Phys (a) -> Boolean;
IsDIMENSION_identifier =
      primitive "IsDIMENSION_identifier";
```

8 where `DIMENSION_identifier` is the name of the dimension. If the dimension name is a basic identifier then the names introduced by these implicit definitions are also basic identifiers. If the dimension name is an escaped identifier then the names introduced by these implicit definitions are escaped identifiers in which the case of each letter that is not dependent upon the dimension name is as shown above. The dimension identifier (escaped or otherwise) is given the suffix `_dim` to denote the general type, and is used alone to denote the type which is particularized with `Float`.

9 The implicit type associated with a dimension is a set of physical quantities of that particular dimension; this type is called a *dimension type*. The polymorphic name in the dimension type corresponds to the numeric part of the physical quantity.

10 The three implicit functions associated with a dimension are primitive functions designed to facilitate the interaction between values of type `Phys` and values of the dimension type. The implicit coercion functions enable the conversion of physical quantities of the dimension type to the type `Phys`, and vice versa. The remaining function returns a boolean value indicating whether or not a value of type `Phys` has a dimensionality that conforms to the defined dimension.

4.2.12.2 Unit Definition

1 A unit definition defines a physical unit in MHDL.

Example:

```
-- a base unit definition
unit meter of length
     variations m, metre, meters, metres;

-- a derived unit definition
unit newton of force = 'kg m / sec^2'
     variations newtons, n;
```

Grammar Rules:

```
unit_definition ::=
    unit UNIT_identifier of DIMENSION_name
        [ = UNIT_designation ]
        [ unit_conversion ]
        [ UNIT_variations_clause ]

unit_conversion ::=
    conversion conversion_spec UNIT_name

conversion_spec ::=
    RATIONAL_expression
  | FUNCTION_expression inverse FUNCTION_expression

variations_clause ::=
```

variations `identifier { , identifier }`

2 A *unit* is a unit of physical measurement that applies to a specific physical property, or dimension; it can be used to construct a physical value for a measurement involving that dimension. A unit definition that has no unit designation or unit conversion is a *basic unit*; a unit that has either a unit designation or a unit conversion is a *derived unit*. Each dimension may have only one basic unit.

3 The basic unit of a basic dimension identifies the primary unit for that dimension; all other units for the dimension are based on this primary unit. The basic unit for a derived dimension is determined as though a unit definition existed for the dimension with a unit designation derived from the dimension designation by replacing each dimension name in the dimension designation with the name of the basic unit for that dimension. A unit definition with a designation equivalent to the implicit designation just defined gives a name to the basic unit using a derived designation; in either this case, or the case of a basic unit, the unit is called the *base unit* for the dimension.

4 A unit is defined by a unit definition. In addition to introducing the name of the unit, the unit definition provides information needed to perform dimensional analysis and conversions to other units of the same dimension. This information is provided through the unit designation and the unit conversion. These constructs are mutually exclusive.

5 If the unit is a derived unit then the unit definition must either include a unit designation that associates a unit with each dimension named in the dimension designation of the associated dimension, or a unit conversion as discussed below. When a unit designation is used, the appropriate conversion for a derived unit is calculated from the unit designation in a manner consistent with normal mathematical usage and the ISO 1000 standard ([11]). It is an error if any unit name in the unit designation includes a unit conversion in its own unit definition.

6 Conversions between units of a dimension are expressed in terms of the base unit; it is an error if the unit named in a unit conversion has a unit conversion in its unit designation consisting of a pair of function expressions (separated by the keyword `inverse`). If the conversion spec is a pair of function expressions, the first takes a physical quantity expressed in the base unit and produces a physical quantity in the defined unit, and the second is the inverse of the first; it is erroneous if the second function expression is not the inverse of the first. If a unit conversion contains a single rational expression, then the expression is a multiplicative factor between the base unit and the unit being defined; this is equivalent to the pair of function expressions `(*) (RATIONAL_expression) inverse (/) (RATIONAL_expression)`.

7 A variations clause introduces a set of identifiers representing variations on the *primary name* introduced by the enclosing unit definition or multiplier specification. The names in the variations clause serve as substitutes, or aliases, of the unit or multiplier name. Use of any name in the variations clause is the same as the use of the primary name.

8 Each unit definition creates a family of unit names created by prefixing each unit identifier (the primary name and each of its variations) with the appropriate multipliers. See 4.2.12.3 [Multiplier Specification], page 49, for a description of multipliers and how scaled unit names are created.

9 The basic units identified by ISO standard 1000 ([11]) are defined in package Standard; see 10.1.17 [Physical], page 129, for the definition of the associated types and units.

4.2.12.3 Multiplier Specification

Grammar Rules:

```
multiplier_specification ::=
  multiplier MULTIPLIER_identifier
    = INTEGER_OR_FLOAT_literal [ MULTIPLIER_variations_clause ]
```

1 A *multiplier* is a prefix to a unit that scales the prefixed unit.

2 A multiplier specification introduces a set of multipliers that can act as prefixes to physical units; see 4.2.12.2 [Unit Definition], page 47 for a description of the variations clause. Since a multiplier is never used by itself, it does not share a name space with other identifiers. Each multiplier identifier must be distinct from all other multiplier identifiers; however, a multiplier identifier does not have to be distinct from non-multiplier identifiers or from reserved words.

3 The relationship between a multiplier and a unit is unlike any other in MHDL. Scaled unit identifiers are created by combining like unit identifiers and multipliers (and their respective variations). A unit and a multiplier may be combined if they are both basic identifiers or both escaped identifiers; the resulting identifier is the same kind of identifier as its constituents. In the case where both the multiplier and the unit are escaped identifiers, the combined identifier is created by eliminating the pair of back quotes in between the two juxtaposed escaped identifiers, visually forming a single escaped identifier. For example, the following items

```
unit meter of length variations meters, m;
multiplier centi = 1.0e-2 variations c;
```

4 result in the definition of the following units: meter, meters, m, centimeter, centimeters, centim, cmeter, cmeters, cm. Multiplier specifications may only appear in Package Standard. It is an error if more than one multiplier prefix precedes a unit. It is also an error if the same identifier can be formed by more than one multiplier / unit combination, as given above.

5 A multiplier specification associates the identifiers denoting the multiplier with an integral or float scale factor. The scale factor may be used if it is necessary to normalize a physical value in order to perform a numeric operation involving a value measured in a unit scaled with the associated multiplier.

Example:

```
multiplier kilo = 1000 variations k;
distance = 1.4 'km';
```

4.2.13 Signal Type

1 Signal types are functions defined from one physical type to another physical type; more formally, signal types are functions of any type that may be coerced to `(Phys(a) -> Phys(a))`.

2 **Note:**

> Typically, the dimension of the domain of a signal type will be time or frequency; however, there is no language restriction that requires this.

3 There are a number of representations for signals that are defined in MHDL. Each kind of signal representation is useful for different kinds of operations and behavior definitions. For most of the signal representations, the standard floating point numeric operations (corresponding to class `Real`: see 10.1.10 [Real], page 123) are available. Some representations are restricted to ordinary arithmetic operations: addition, subtraction, and multiplication (as defined in class Num: see 10.1.4 [Num], page 121, for a specific definition). Those classes which are so restricted are listed below. The general class of signal definitions available in MHDL are:

- Discrete signals. These signals are represented as lists of discrete values. Asynchronous and synchronous representations are both available. See 10.4.3 [Discrete Signals], page 167, for the exact definition of this class.

- Continuous signals. These signals cover all continuous representations. One of the most important of these representations consists of MHDL functions of the above type: `(Physical -> Physical)`. See 10.4.1 [Universal Signals], page 163, for a definition of this representation. The other portion of the continuous signals are the periodic signals.

- Periodic signals. These signals have an explicit period in the domain unit (usually time). Periodic signals are limited to the functions in class Num. See 10.4.4 [Periodic Signals], page 167, for a definition of these signals.

4 In addition to the standard numeric functions, signals have some additional operations defined on them. Some apply to any signal: for example, any signal may be evaluated at a point in its domain, scaled by a constant factor, or delayed by a constant factor. See 10.4.1 [Universal Signals], page 163, for a definition of these functions. Each of the classes above has additional functions defined on them as well. These functions may be found in their respective definitions sections, listed above.

4.3 Type Coercions

1 Arguments in a function application must agree in type with the type specification in the definition of that function. Where this is not the case, functions may be implicitly inserted automatically to convert the types of the arguments to those in agreement with the type specification for the function. This process is called *type coercion*.

2 Type coercions involve special functions, called *coercion functions*, whose name definitions are prefixed with the keyword *coercion*. A coercion graph is defined whose nodes are the type specifications occurring in any name or abstract property definition in the description, where structurally equivalent types are represented by the same node. A directed edge between types a and b occurs whenever a coercion function is defined whose type is `a -> b`. Such a coercion graph exists for each context in which an expression exists, as a function of the coercion functions visible from that point. It is an error if any of the types in the coercion graph consist solely of a polymorphic name.

3 An expression occurs in a context, which may be a function application or a syntactic construct (such as an if statement or a value definition) which demands an expression of a particular type. In the former case, where a function application has one or more arguments of the wrong type, then the arguments are considered as a set of expressions (called the *base set* of the application). In the latter case, the expression is considered in isolation, and the base set includes the expression itself. An argument of type `a` may be coerced to type `b` if there is a directed path from `a` to `b` in the coercion graph; the argument is changed by inserting an application of each coercion function that created a directed edge when that edge is traversed in the path from `a` to `b`. The changed argument then has type `b`, and is said to have been coerced to type `b`.

4 A set of expressions `s` is *less coerced* than a set of expressions `t` if `s` may be coerced into `t` by the insertion of at least one coercion function. A set of expressions `s` is *minimally coerced* with respect to a set of required types if the types of the expressions agree with that required by the set of required types and for any other set of expressions `t` coerced

from the same base set that also agrees with that required by the set of required types, `t` is not less coerced than `s`. Any minimally coerced set of expressions may be used in the description unit where a set of required types may be determined, such as in a value definition (where the set of required types is determined by the name definition), or a function application (where the set of required types is determined by the type of the function being applied); however, only minimally coerced expression sets may be used. It is erroneous if the results of using a given minimally coerced set of expressions is different from using another, different minimally coerced set of expressions.

5 A set of expressions (of possibly differing types) may be coerced to the same type by the insertion of a series of type coercion functions around the expressions. Such a type is said to be an *upper type bound* of the set of expressions. Given two upper type bounds of a set of expressions, if expressions of one type may be coerced to the other type it is said to be less coerced than the other, as above. A type `a` is said to be a *minimal upper type bound* if, given any other upper type bound `b`, either `a` is less coerced than `b` or `a` cannot be coerced to `b`.

6 A variety of type errors may occur, as discussed earlier in this chapter. In the event of a type error, coercion functions may be automatically inserted into the expression in which the type error occurs. This process occurs in two steps.

7 In the first step, a set of sets of related expressions, which are subexpressions of the source expression, is determined as given below. A minimal upper type bound is determined for each set of related expressions. Type coercion functions are then automatically inserted as follows:

8 In the second step, there is a disagreement between a desired type and the inferred type of an expression. The desired type may be given by a name definition, the type on the left hand side of a value definition, the type of a function, (where the inferred type is the type inferred for the argument), or an if or case statement, where the if or case expression is viewed as the application of an if or case function to the subexpressions of the expression (note that the term function above includes constructors, such as the tuple and list constructor). Sets of related expressions are formed by the expression itself if it is not a function application, or by the arguments of the top level function application and the function application itself if it is a function application. Sets of related expressions are determined by the desired type applying to the argument or result; one for each polymorphic name in that type that applies to one of the function arguments, and one for each type that is not a polymorphic name. Type coercion functions are inserted to convert each argument of the function to a minimal upper type bound for the set of related expressions in which it is contained, or to the (unique) type given in the name definition if that type is not a polymorphic name. If the resulting function application does not match the context of the desired type, other minimal upper type bounds of the sets of related expressions are tried (if they exist). It is an error if no minimal upper type bound is found such that the resulting coerced function

application matches the context of the desired type.

9 The insertion of coercion functions in a given expression in a package or model is not affected by the inclusion of that package or model in the inclusion list of another package or model. In other words, the insertion of coercion functions is determined, according to the rules above, within the visibility map of the package or model where the expression occurs, and not the including package or model.

Chapter 5

1 # Properties

2 The meaning of an MHDL description is encapsulated in the values that exist in various parts of the description. These values are identified and read using names, and are thus called *named values*. Many of these values are specifically attached to names in the description, particularly structure names and component names. These values are called *properties*. Values are identified as existing with name definitions, and are specified with value definitions. See 6.2.1.1 [Functions], page 67, for the syntax of these two constructs.

3 There are two kinds of properties: attributes and signals. An attribute is a piece of design information that is attached to a particular piece of the design. Examples are design parameters such as resistance or power consumption. A signal is a representation of the information that occurs at a specific place in the design, and which usually has a relationship to a physical location in the final designed analog or microwave hardware. Signals are usually of a signal type (see 10.4 [Signal Types], page 163, for more information).

4 Properties are attached to names in the design hierarchy. In some cases, properties attached to one name may be read through another name, as if the property were also attached to the second name. This is accomplished by associations. Connectors in particular are associated by connect specifications. See 3.1.2 [Connectors], page 21 for a full discussion of structures and connectors, and how they are associated.

5.1 Abstract Properties

1 Abstract properties identify names used for values at many points in an MHDL description.

Example:

```
attribute Input_Value, Output_Value : Phys (Float);
```

Grammar Rules:

```
abstract_property_definition ::=
    property_kind abstract_property_name_list : type_specification

property_kind ::=
    attribute
  | signal

abstract_property_name_list ::=
    ABSTRACT_PROPERTY_simple_name
        { , ABSTRACT_PROPERTY_simple_name }
```

2 Properties are values which are attached to unique names in the design hierarchy. An *abstract property* is the identification of a name by which a value may be attached to another name. This name is given a type, and all properties having that simple name will be a member of the set of values defined by the type.

3 Properties are always attached to unique names in the design hierarchy; thus, each property may be thought of as existing in a specific part. It is illegal to attach a property to a name in a model or package using a qualified name. See 8.3 [Qualified Names], page 101, for a definition of qualified names.

4 A Property may be attached to a name which indicates any of the following items in a design hierarchy:

- A model, in which case the property is attached to the name of each part that was created by the model thus named.
- A package, in which case the property is attached to the name of each portion of the design unit created from that MHDL definition that included the package definition: a model, structure, or component definition.
- Another property.
- A component.
- A structure.
- A connector.

- A label associated with a value definition or an equation.
- A constraint.
- A connection, via the connection label.

5 Properties may be attached to a name by one of three mechanisms:

- A default value definition (see 6.2.2 [Pattern Bindings], page 69, for more information).
- An equation that uses the property in the equation (see 6.5 [Equations], page 76, for more information). This is called an equation definition for the property on that name.
- A normal (non-default) value definition (see 6.2.2 [Pattern Bindings], page 69, for more information).

5.2 Property Value Definitions

1 A property value definition gives a value to a property.

Example:

```
out.Voltage = (magnitude * sin(frequency * t)) 'volts';
```

2 A *property value definition* is a definition that specifies values for a collection of properties. See 6.2 [Value Definitions], page 63, for the syntax associated with value definitions in general. A property value definition is identified by the name of a property as the value definition target (see 8.4 [Hierarchical Names], page 102, for a definition of property names). A property value definition is also known as an *assignment*. The value in the assignment must be in agreement with the type in the abstract property definition with the same abstract property name as the assignment; it is an error if no such abstract property definition exists. Note that type coercion does apply to assignments; see 4.3 [Type Coercions], page 51, for more details. Whenever the property name is used in other definitions or expressions, the value given by the assignment is used.

5.3 Associations

Grammar Rules:

```
association_definition ::=
    associate name , name { , name }
```

1 If a set of names are *associated*, then the properties attached to one name may be read as if they were attached to any of the associated names as well, under certain conditions; in this case, the value of the property is said to be read through the associated name. In addition, associations may affect how and whether properties are attached to the associated name. Properties are attached to associated names, and read by other names with which the name is associated, according to the following rules. For a given property, a given name, and set of other names with which it is associated:

1. If the name has a default value definition for the property, and no other associated name has a default value definition, an equation definition, or a non-default value definition for that property, then the property is attached to the given name with the value in the default value definition, and that value may be read through all the other associated names.

2. If more than one of the names has a default value definition for the property, and none of the associated names has an equation definition or a non-default value definition for the property, then the property is attached to the names for which a default value definition exists using the value in the respective default value definitions. Those names without a default value definition cannot read the value of the property through the associated name.

3. Similarly, if the name has an equation definition, and no other associated name has an equation value definition or a non-default value definition, then the property is attached to the given name with the value determined by the equation definition (see 6.5 [Equations], page 76, for more details). That value may be read through all the other associated names.

4. If more than one of the names have equation definitions, and none of the associated names has a non-default value definition for the property, then the property is attached to those names for which an equation definition exists with the values determined by the respective equation definitions. Those names without an equation definition may read a value given by a default value definition if one exists; if one does not exist, then the value of the property cannot be read through the associated name.

5. If one of the names has a non-default value definition, and no other name has a non-default value definition, then the property is attached to the name with the value definition with the value specified in the value definition, and that value may be read through the other associated names.

6. If more than one of the names has a non-default value definition, then the property is attached to those names which have non-default value definition using the value in the respective value definition. Names without a non-default value definition may have the value of an equation or a default value definition (if no equation value definition exists) given to the property attached to the name; if no such definition exists, the property cannot be read through the associated names.

2 For any property name that is either attached to or read through two of the names in the set of associated names, the types defined by the abstract property definitions of those names in the two references must be the same.

3 Names are associated if they occur in the same association definition, if they occur in the same connection list (if they are connectors), if they are the names of a component and the structure indicated by that component, or if they are both associated with another name. See 3.1 [Structure Definitions], page 17 for more details concerning the semantics of structures, connection lists, and components.

4 **Note:**

> The value of attributes of the same name attached to associated names may still be read using one of the structural functions. See 10.5.4 [Structural Functions], page 172, for more details.

5 An association definition explicitly gives all the names to be associated.

Chapter 6

Named Values

An item with a name that represents a value is called a *named value* in MHDL.

As explained elsewhere (see 5 [Properties], page 55), named values may be attached to other language items in which case they are called properties. Properties are of particular interest in MHDL due to the role they play in connection semantics (see 3.1.2 [Connectors], page 21). Occasionally the term *variable* is used for a named value that is not a property.

There exist items in MHDL that are named but that do not represent values. That a named value represents a value implies, among other things, that every named value has a type associated with it. As an example, the name of a connector and the name of a model are not named values since these items do not have types in MHDL. Abstract properties are other examples of named items in MHDL that are not named values. A particular "instance" of an abstract property as a property certainly represents a value, but the abstract property on its own does not (see 5.1 [Abstract Properties], page 55).

Expressions in MHDL can be thought of as performing operations on named values and producing as a result anonymous values. For example, an operator may be applied to two named values yielding an unnamed value that is the result of the operation. See 7 [Expressions], page 79 for a complete description of MHDL expressions.

A fundamental characteristic of a named value is that it "takes on" a value through a process called *binding*. Binding a named value to a value takes place as a basic part of *pattern matching* in MHDL. The pattern-matching semantics of MHDL is central to the language since pattern matching takes place in a wide variety of contexts. For example, in a case expression, pattern matching is used to select among a set of alternatives in computing a value. As another illustration, in a function application, pattern matching is used to associate the argument values with the variables used to express the function

result. See 7 [Expressions], page 79 for a description of these constructs.

7 Pattern matching also takes place in an MHDL construct whose sole purpose is to bind named values to values. This construct is called a *value definition* (see 6.2 [Value Definitions], page 63 for a complete description). Named values that are bound by value definitions may be explicitly typed by a *name definition*. A name definition does not define a value for a named value, it merely specifies the name and associated type of the named value. See 6.1 [Name Definitions], page 62 for a complete description of MHDL name definitions.

8 A particularly important kind of named value in MHDL is a named value of a function type (see 4.2.3 [Function Types], page 41). Such a named value is called a *function*. By defining a mapping of "inputs" to "outputs", MHDL functions are the basic building blocks for behavioral descriptions expressed in the language.

9 The final concepts of key relevance to named values in MHDL are *constraints* and *equations*. Intuitively, a constraint is a check to be performed (or an assertion to be tested) concerning values of named values while an equation can be thought of as expressing a criterion for an acceptable set of values for named values.

6.1 Name Definitions

1 A name definition is a construct in MHDL for specifying the name and associated type of a named value.

Grammar Rules:

```
name_definition ::=
    [ coercion ] named_values :: type_specification
        [ where context ]

named_values ::=
    NAMED_VALUE_simple_name {, NAMED_VALUE_simple_name }
```

2 A name definition consisting of multiple named values is equivalent to a set of name definitions, one per named value, with the *coercion* keyword (if present), the context (if present) and type designation of the original name definition duplicated in each name definition in the set. Only one name definition is allowed per named value.

3 A named value that appears in a name definition must be given a value by one or more contiguous value definitions in the same definition list as the name definition (after generate definitions have been elaborated). The type of the expression specified in the

name definition must be a specialization of the type of the value definition. If no name definition is used for a named value, all value definitions for the named value must be contiguous and appear in the same definition list.

4 Similar to typed expressions (see 7.7 [Typed Expressions], page 87), the type indicated in a name definition for a named value must be a specialization (see 4 [Types and Classes], page 29) of the type of the value bound to the named value.

5 A name definition including the keyword *coercion* indicates that the named values are to be interpreted as type coercion functions. In this case, the type designation in the name definition must be a function type. See 4.3 [Type Coercions], page 51 for more information on type coercion.

6 See 4.3 [Type Coercions], page 51 for a description of the semantics of type coercion functions. See 4 [Types and Classes], page 29 for the definitions of a context and a type designation.

6.2 Value Definitions

1 A value definition is an MHDL construct whose purpose is to identify the value which is represented by the name of a named value.

Grammar Rules:

```
value_definition ::=
    [ label ] value_definition_target = expression
        [ where_clause ]
  | [ label ] value_definition_target guarded_expression_list
        [ where_clause ]

value_definition_target ::=
    function_designation
  | [ default ] pattern

function_designation ::=
    FUNCTION_name ( pattern { , pattern } )
  | pattern operator pattern

guarded_expression_list ::=
    guard = expression [ guarded_expression_list ]

where_clause ::=
```

where `nested_definitions`

2 Value definitions fall into two categories depending which of the two forms the value definition target takes. If the value definition target is a function designation, the value definition is called a *function binding*; otherwise, the value definition target is a pattern (optionally preceded by *default*) and the value definition is termed a *pattern binding*.

3 Certain value definitions are ambiguous syntactically as to whether they represent function bindings or pattern bindings. These ambiguous value definitions are possible because value definition targets such as `x(y)` can be interpreted as a function designation or a pattern depending on whether `x` is considered a constructor or a named value. The ambiguity is resolved in MHDL by the rule that says `x` in such a value definition target is considered as the constructor `x` if the value definition is within the scope of a constructor `x`; otherwise, `x` is considered as a named value. A similar ambiguity exists if the function designator is in the form of an infix operator and is resolved in the analogous way; for example, `x op y = z` is interpreted as a function binding unless `op` is a constructor operator in the current scope in which case it is a pattern binding. See 4 [Types and Classes], page 29 for a full discussion of MHDL constructors.

4 Value definitions have types associated with them. If the value definition does not include a guarded expression list, the type of the value definition is the type of the expression in the value definition. If the value definition includes a guarded expression list, each expression immediately following `=` symbols in the guarded expression list must be of the same type and this common type is the type of the value definition.

5 A value definition containing a guarded expression list sequence is called a *guarded value definition*. The guard in a guarded expression list must be an expression of type `Bool`. The textual order in which the guarded expression list constructs appear is significant. The expression that defines the value used in the binding is the one corresponding to the first guard that evaluates to `True`. If no guard evaluates to `True` the value used in the binding is $\perp$. The variable `otherwise` is defined to `True` in the MHDL package Standard for convenience in writing the last alternative in a guarded expression list sequence.

6 A non-guarded value definition is equivalent to a guarded value definition containing a single guard whose value is `True` and a single expression that is identical to that in the non-guarded value definition.

7 A value definition containing a where clause includes locally scoped definitions that only apply within the value definition. As with the nested definitions in let expressions, the nested definitions in a value definition may be mutually recursive and pattern bindings in these nested definitions are matched lazily (see 7.4 [Let Expressions], page 84).

8 See 7.6 [Case Expressions], page 85, for the definition of a guard. See 6.3 [Patterns], page 72, for the definition of simple and general patterns. See 7.4 [Let Expressions],

page 84, for the definition of nested definitions.

9 The next two subsections consider in detail the two kinds of value definitions possible in MHDL.

6.2.1 Function Bindings

1 As explained above (see 6.2 [Value Definitions], page 63), a function binding is one of the two kinds of value definitions in MHDL.

2 In MHDL the term *function* as a noun is used to mean "a named value of a function type". Function bindings are used in MHDL as a convenience in defining functions. Put another way, function bindings tie functions to particular function values.

3 A function binding is said to *apply to* a single function. A function binding applies to the function indicated by the function name or operator in the function designation of the function binding.

4 The order in which function bindings textually appear in a list of definitions in a description unit is significant, as is the "precedes" operation between definitions in different description units. For any application of a function, the function binding that first pattern-matches successfully with respect to the arguments is the binding that defines the value of the function, where the term "first" means that definition which precedes or textually is before the other matching definitions; it is an error if this relationship is ambiguous. See 6.3 [Patterns], page 72 for a description of pattern matching.

5 Each function binding has an *arity* associated with it. The arity of a function binding is the number of patterns in the function designation. All of the function bindings that apply to a function must have the same arity. This common arity is called the arity of the function.

6 All of the function bindings that apply to a function must have the same type (see 6.2 [Value Definitions], page 63 for a description of the type of a value definition).

7 The patterns appearing in the function designation must be *linear*. That is, no variable may appear more than once in this collection of patterns.

8 A function binding may bind a function value to either a function or an operator, depending on which of the two forms of the function designation is used.

9 Like all named values, a function may be explicitly typed with a name definition. In this case the stated type of the function must be a specialization (see 4 [Types and Classes], page 29) of the type of any function bindings that apply to the function.

10 There exist formal equivalences that may clarify the relationship of function bindings to other aspects of the language. Two of these equivalences are explained here. First, the

semantics of "selecting" a function binding to use when applying a function is identical to that of "selecting" an alternative in a case expression (see 7.6 [Case Expressions], page 85). More precisely, suppose the complete set of function bindings that apply to a function `f` is:

```
f(p11, p12, ..., p1k) match1
f(p21, p22, ..., p2k) match2
...
f(pn1, pn2, ..., pnk) matchn
```

11 where the `pij` are patterns and the `matchi` are the sequences of text that follow the function designations in the function bindings. This set of function bindings is semantically equivalent to

```
f(x1, x2, ..., xk) = case (x1, x2, ..., xk)
                of {(p11, p12, ..., p1k) match1
                    (p21, p22, ..., p2k) match2
                    ...
                    (pn1, pn2, ..., pnk) matchn
                   }
```

12 where the `xj` are new identifiers introduced to not conflict with existing identifiers. Thus regardless, of how a function is defined, it may be specified equivalently with a single function binding. A second equivalence that should be observed is one between function bindings and pattern bindings based on the use of lambda abstractions. Continuing with the function `f`, the definition for `f` immediately above is equivalent to the following pattern binding:

```
f = \(x1, x2, ..., xk) -> case (x1, x2, ..., xk)
                of {(p11, p12, ..., p1k) match1
                    (p21, p22, ..., p2k) match2
                    ...
                    (pn1, pn2, ..., pnk) matchn
                   }
```

13 The combination of the two equivalences described here imply that any function, no matter how specified, may always be defined equivalently with a single pattern binding.

Example:

```
f(x, y) = (x + y) / 2
   -- Defines a two-argument function f that returns
   -- the average of its arguments.  Identical to
   -- f = \ (x,y) -> (x + y) / 2.

nonnegative(x) | x < 0     = 0
               | otherwise = x
   -- Defines a single argument function nonnegative using
   -- a guarded function binding.

1 + y = g(y)
x + y = h(x, y)
   -- This pair of function bindings define the
   -- operator (+).  If the left argument has the value 1,
   -- the first rule defines the result of the operator;
   -- otherwise, the second rule defines the operator value.
```

6.2.1.1 Properties of Function Values and Types

1 *Functions* are important building-blocks of an MHDL description. As explained above, functions are usually defined with the function-binding language mechanism. However, functions may also be specified using pattern bindings (see 6.2.2 [Pattern Bindings], page 69) as well as anonymously using lambda abstractions (see 7.3 [Lambda Abstractions], page 83).

2 An MHDL function is a form of computation that takes as inputs one or more input *arguments* and returns a resulting value. These arguments correspond to the patterns that appear in a function designation or in a lambda abstraction. As explained above, the number of these patterns is the arity of the function.

3 Functions may be *applied* (i.e., used) freely within MHDL expressions (see 7.1 [Function Applications], page 81).

4 Each MHDL function has a type associated with it; the type of a function specifies the allowed types of arguments and the possible types of the returned results. As explained above, the type of a function may be stated explicitly with a name definition or may be inferred from the value binding for the function.

5 Function types are "first class" in the language; that is, properties may be defined to be of a function type, function values may be returned from functions, etc.

6 MHDL includes the concept of an *infix operator*; an infix operator is a function that is invoked using an operator symbol in infix notation (see 7.2 [Operator Applications],

page 82).

7 It is common in MHDL to define a function with irrefutable patterns that represent the function arguments (see 6.3 [Patterns], page 72 for a discussion of irrefutable patterns). Such functions evaluate their arguments "lazily" (or only "as needed") during the course of applying the function.

8 An MHDL function may be *partially applied* by omitting certain arguments when invoking the function. This process, known as *currying*, returns a function which, in an intuitive sense, can "complete" the computation when supplied with values for the omitted arguments.

9 The basic form of a function type in MHDL is

10
```
t1 -> t2
```

11 where `t1` and `t2` are any types. Here, `t1` and `t2` are the *argument type* and *result type* of the function, respectively. If a function `f` has argument type `t1` and result type `t2`, for example if there is a name definition for `f` such as

12
```
f :: t1 -> t2
```

13 and if `e1` is an expression of type `t1`, then

14
```
f(e1)
```

15 is an application of `f` with the argument `e1` and results in a value of type `t2`.

16 What can be thought of as "multi-argument" function types are possible in MHDL as follows:

17
```
t1 -> t2 -> ... -> tn
```

18 where `t1`, `t2`, ..., `tn-1` are the argument types and `tn` is the result type. Due to the associativity of "`->`", this function type is equivalent to

19
```
t1 -> (t2 -> (... -> tn) ... ).
```

20 Putting this relationship another way, an n-argument function in MHDL is always equivalent to a 1-argument function whose result type is the n-1-argument function type that can be used to complete the invocation of the original function by accepting the remaining arguments. If there exists a name definition for `f` such as

21
```
f :: t1 -> (t2 -> (... -> tn) ... )
```

and `e1`, `e2`, ..., `en-1` are of type `t1`, `t2`, ..., `tn-1`, respectively, then an application of f may be written

```
f(e1, e2, ..., en-1).
```

Due to the associativity of function application this is equivalent to

```
( ... (f(e1))(e2)) ... )(en-1).
```

The function application `f(e1)` to the multi-argument function `f` can be thought of as a "projection" of the function f in the sense that it "fixes" certain arguments (namely the first one) and allows others to remain "free" (namely all but the first). To understand this general capability in MHDL, consider a function `f` whose name definition is as follows:

```
f :: t1 -> t2 -> t3 -> t4
```

If `e1`, `e2`, `e3` are of types `t1`, `t2`, and `t3`, respectively, the following are some example applications of this function `f`:

```
f(e1)       -- Binds t1 argument, others are free
            -- Result type: t2 -> t3 -> t4

f(e1,,)     -- Same as above, omitted arguments may
            --     be shown as being explicitly absent

f(,e2)      -- Binds t2 argument, others are free
            -- Result type: t1 -> t3 -> t4

f(e1,,e3)   -- Binds t1 and t3 arguments, others are free
            -- Result type: t2 -> t4

f(e1,e2,e3) -- Binds each argument
            -- Result type: t4
```

6.2.2 Pattern Bindings

As explained above (see 6.2 [Value Definitions], page 63), a pattern binding is one of the two kinds of value definitions in MHDL. A pattern binding is the more basic and general means for binding named values to values. As explained above (see 6.2.1

[Function Bindings], page 65), the other form of value definition, a function binding, provides a convenient manner for specifying function values but can be thought of as "syntactic sugar" with semantics that can be defined in terms of pattern bindings.

2 The pattern in a pattern binding is treated as an irrefutable pattern by default (see 6.3 [Patterns], page 72 for a discussion of irrefutable patterns). That is, it is as though the pattern were surrounded by parentheses with the parenthesized pattern preceded by `~`. Due to this treatment of the pattern as an irrefutable pattern we say that pattern bindings are matched *lazily* and only when the named values in the pattern are read. See 6.3 [Patterns], page 72 for a complete description of pattern matching in MHDL.

3 A pattern binding that includes the keyword *default* is called a *default pattern binding*. A default pattern binding may not appear as a definition in a let clause or in a where clause. Default pattern bindings allow "tentative" bindings to be made to named values that may be "overridden" with a subsequent pattern binding (typically in a deriving or inheriting model or package). See discussion of value bindings, below, for details on this capability.

4 The first form of a pattern binding given above without the optional *where* clause (i.e., the form `p = e` where `p` is a pattern and `e` is an expression) is called a *simple pattern binding*. The general (guarded) form of the pattern binding:

```
p | g1 = e1
  | g2 = e2
   ...
  | gn = en
  where defns
```

5 (where `p` is a pattern, the `gi` are boolean expressions, the `ei` are expressions, and `defns` represents nested definitions) is semantically equivalent to the simple pattern binding:

```
p = let defns in
      if g1 then e1 else
      if g2 then e2 else
      ...
      if gn then en else
      error("Unmatched pattern")
```

6 A pattern binding provides a *value binding* for each named value that appears in the pattern. More precisely, these value bindings are those obtained by irrefutably matching the pattern against the expression in the simple-pattern-binding equivalent (see above) of the pattern binding. In a similar way, the set of function bindings that apply to some function (see 6.2.1 [Function Bindings], page 65) constitute a single value binding for the function. It is the value bindings for a named value that determine the

value of the named value. In particular, for any set of definitions that make up a part (see 3.3 [Design Hierarchies], page 25 for the definition of a part), a named value may have a number of default value definitions and either zero or one value definitions that are not default value definitions. If all the value definitions are default value definitions, then that value definition that is preceded by all other value definitions is used to provide the value of the named value, and all other value definitions are ignored; it is an error if such a default value definition does not exist (see 2.1 [Inclusion Lists], page 7, for the definition of the precedes relation). In addition, if all value definitions are default value definitions, a value may also be provided for the named value that agrees with all the equations and constraints of the description by a mechanism which is defined by the implementation; in this case, all default value definitions are ignored. If a value definition that is not a default value definition exists, it is used to provide the value of the named value. These rules may be changed by the presence of associations; see 5.3 [Associations], page 58, for more information.

Example:

```
otherwise = TRUE
   -- A pattern binding which binds named
   -- value otherwise to the value TRUE.

(x,y,z) = my_tuple
   -- In this example, my_tuple must be a three-tuple; the
   -- pattern binding binds x, y, and z to the first, second
   -- and third components of my_tuple, respectively.

head(list) = first where { (first:-rest) = list }
   -- This is a function binding.  However, the definition
   -- in the WHERE clause is a pattern binding that is used
   -- in expressing the function result.

double = \ x -> x + x
ten = double(5)
   -- A pair of pattern bindings with a use of the named
   -- value bound in the first binding in the second
   -- binding.

trouble = 1/0
   -- Don't try to read trouble or you'll get ⊥.
```

6.3 Patterns and Pattern Matching

1 Pattern matching is a fundamental MHDL concept and takes place in a variety of language contexts. In addition to affecting what may be intuitively thought of as "flow of control" or "selection from a set of alternatives", the pattern-matching process is what is used in MHDL to bind named values to values.

Grammar Rules:

```
pattern ::=
    pattern [ CONSTRUCTOR_operator pattern ]
  | - integer_or_float
  | simple_pattern
  | constructor ( pattern { , pattern } )

integer_or_float ::=
    integer_literal
   |   floating_point_literal

simple_pattern ::=
    NAMED_VALUE_variable
  | as_pattern
  | constructor
  | literal
  | wildcard_pattern
  | null_pattern
  | ( pattern )
  | tuple_pattern
  | list_pattern
  | explicit_irrefutable_pattern

as_pattern ::=
    variable @ simple_pattern

variable ::=
    name

wildcard_pattern ::=
    _

null_pattern ::=
```

```
    ()

tuple_pattern ::=
    ( pattern , pattern {, pattern } )

list_pattern ::=
    '[' [ pattern {, pattern } ] ']'

explicit_irrefutable_pattern ::=
    ~ simple_pattern

constructor ::=
    name
```

2 Pattern matching in MHDL takes place with respect to a *pattern* and a *value*. In particular, the pattern is being *matched against* the value. Three outcomes are possible when matching a pattern against a value.

1. The pattern matching may *fail*.
2. The pattern matching may *succeed*. In this case, a side effect of the successful pattern match is a binding of each named value in the pattern to a value.
3. The pattern matching may *diverge*. Conceptually, in this case the pattern-matching process returns $\perp$.

3 Any pattern in MHDL must be *linear*. A linear pattern is one with no named value appearing more than once in the pattern. Intuitively, this rule insures that there is no ambiguity should pattern matching succeed and named values in the pattern be bound to values.

4 In the following discussion of patterns and pattern matching, examples are given in terms of function applications but the pattern-matching effect is the same regardless of the context in which it occurs.

5 An *as pattern* is of the form `var@pat` where `var` is a named value and `pat` is a simple pattern. An as pattern matches against a value if and only if `pat` matches against the value. If it does match, `var` will be bound to this value. As patterns are useful in "decomposing" a value while still retaining a name for the whole. For example:

```
f(x@(y,z)) = ...
```

6 defines a function that can be applied with a two-tuple for the argument. In this case, `x` will be bound to the two-tuple, and `y` and `z` will be bound to the first and second components of the two-tuple, respectively.

7 A *wildcard pattern* matches any value. For example:

```
f([_, x, _]) = ...
```

8 defines a function that can be applied with a list of length three as an argument in which case x will be bound to the second component of the list. The use of wildcards here documents the fact that the function does not depend on the values of the first and third list elements.

9 A *constructed pattern* is one created by constructors (including constructor operators), tuple patterns, and list patterns. String literals are also constructed patterns due to their definition as lists of characters; likewise character literals, numeric literals, and the null pattern are constructed patterns due to their treatment as "nullary constructors" (matching of numeric literals makes use of the `==` operator).

10 An *irrefutable pattern* is one that always matches. For example, a pattern consisting only of the name of a named value is an irrefutable pattern. Patterns that are normally not irrefutable can be made irrefutable with an explicit irrefutable pattern (a simple pattern prefixed by the `~` symbol). Besides names of named values, the other patterns that are implicitly irrefutable consist of wildcard patterns and as patterns where the simple pattern is irrefutable (i.e., `var@pat` is irrefutable when `pat` irrefutable). A pattern that is not an irrefutable pattern is a *refutable* pattern.

11 In a conceptual sense, when an irrefutable pattern is matched against a value, the match always succeeds, and the named values in the irrefutable pattern are bound to the values they would normally be bound to if the pattern were not irrefutable and the match succeeded; otherwise, these named values are bound to $\perp$. In an operational sense, the value against which an irrefutable pattern is being matched is not evaluated until the named values in the pattern are read and their bound value is required. If at this time, the pattern matching succeeds, the named values are bound as normal; if it fails, the named values are bound to $\perp$. Due to this "delayed" evaluation of the value which is being matched against, irrefutable patterns are said to be *matched lazily*, i.e, "only as needed". Similarly, the value against which the irrefutable pattern is being matched is *lazily evaluated*, i.e., the value of the expression being matched against is actually not evaluated or computed until it is needed for the delayed binding of named values to values.

12 Several examples may make this effect clear. Consider the following definition for a function `f`:

```
f(~(x:-xs)) = xs
```

13 The pattern used in this value definition is an explicit irrefutable pattern. Applying `f` with any list expression as an argument will match. When `xs` is read on the right-hand-side of the `=` symbol, the argument will be evaluated and `x:-xs` will be matched

against this value. If this value is a non-empty list, `x` and `xs` will be bound to the head and tail of this list value, respectively; otherwise `x` and `xs` will both be bound to $\perp$. Consider another value definition for `f`:

```
f(x) = 0
```

14 The pattern used here is (implicitly) also an irrefutable pattern. Applying `f` passing any expression as an argument will match. Since `x` is never read on the right-hand-side of the `=` symbol, the argument will never be evaluated. Put another way, the argument is not evaluated in this case because its value was "not needed". This is why it is sometimes said that arguments to functions in MHDL are lazily evaluated. This behavior is a consequence of the use of irrefutable patterns in function bindings. This is not only an issue of efficient execution but one of semantics as well. For example, if `f` immediately above was applied with a non-terminating computation for its argument, it is not an "error" and the value returned from the function is `0`.

15 The general MHDL rules for pattern matching a pattern against a value are as follows:

1. If the pattern is irrefutable, pattern matching succeeds and the named values are bound as described above.
2. If the pattern is refutable and the value is $\perp$, then the pattern matching diverges.
3. If the pattern is a refutable constructor pattern and the value is not $\perp$, then the pattern matching fails if the outermost constructors are different. Otherwise (the outermost constructors are the same) the result of the match is the result of matching the sub-patterns in a left-to-right manner. Specifically, if all these sub-matches succeed, the overall match succeeds; the first sub-match to fail or diverge causes the overall match to fail or diverge, respectively.
4. Finally, the effect of matching an as pattern `var@pat` against a value is the same as that of matching `pat` against the value with the additional effect that `var` is bound to the value.

16 When matching a pattern against a value, the type of the pattern and that of the value have to be compatible in the usual sense. Additional specific type-compatibility rules are that an integer literal pattern and a floating literal pattern may only be matched against a value in `Num`, `Fractional`, and `Integral`. The number of subpatterns provided in a pattern match using a constructor must equal the arity of the constructor.

6.4 Constraints

1 A constraint specifies an assertion to be tested concerning named values in an MHDL description.

Example:

```
constraint Low_Threshold_Units : Is_voltage(V)
  report "Error: unit of Low threshold for V is not Voltage";
```

Grammar Rules:

```
constraint_definition ::=
    constraint CONSTRAINT_simple_name : BOOLEAN_expression
        [ report STRING_expression ]
```

2 The boolean expression in a constraint definition is called a *constraint*. A constraint represents a condition which should be true at some point during the evaluation of the values in the description. The relationship between the determination of the values in the description and the evaluation of constraints is implementation dependent; however, in order for the named values in the description to be considered correct, all constraints must be met (i.e., all boolean expressions in the constraints must evaluate to `True`).

3 A constraint may have an optional expression of type `String` associated with it. This expression may serve as an optional error message; the use that is made of this expression is implementation defined. In addition, the time of evaluation of the boolean expression, and the display or printing of the error message to the user is implementation defined; however, when the error message is communicated to the user, it must include the full hierarchical name of the constraint that is violated.

4 **Note:**

> The evaluation of constraints may be appropriate at different times in different analysis methods and cases. The implementor is encouraged to allow the evaluation of constraints upon user input.

6.5 Equations

1 An equation in MHDL specifies a required relationship between named values.

Example:

```
R1: v == i * resis ;
```

Grammar Rules:

```
equation_definition ::=
    [ label ] expression

label ::=
    identifier ':'
```

2 The expression in an equation definition is called an *equation*. The purpose of an equation is to help define or bound the named values referenced within the equation. The equations in an MHDL description may be used by a tool to determine acceptable values for the named values in the description.

3 The expression in an equation definition must either be of type `Bool` or of a function type whose result type is `Bool`. In order for the named values in a description to be considered *correct*, all equations in the description must be *satisfied*. An equation of type `Bool` is satisfied if it evaluates to `True` given the values for the named values in the description. An equation of a function type whose result type is `Bool` is satisfied if, for each value in the argument type of the function, the function returns `True` given that value as well as the values for the named values in the description.

4 **Note:**

> Equations will frequently be extracted and given to a tool for evaluation. The predefined function `equations` will often be used to collect equations for solution by an external tool, and the results of the tool assigned to the specific properties using assignments.

Chapter 7

1 # Expressions

2 Expressions in MHDL are a form of computation that results in a *value*. Every MHDL expression has a *principal type* associated with it that must match the type required for the context in which the expression appears. Conceptually, expressions that result in errors when evaluated or whose evaluation does not terminate result in a distinguished value (shared by all types) written $\perp$ and pronounced "bot".

Grammar Rules:

```
expression ::=
    function_application
  | operator_application
  | lambda_abstraction
  | let_expression
  | conditional_expression
  | case_expression
  | typed_expression
  | structure_function_creation
  | derivative_or_integral
  | simple_expression
```

3 In the sections that follow syntax and semantics are given for the various operators that can be used in MHDL to build expressions. To enhance the presentation, issues of precedence and associativity are generally omitted in these sections and the general operator precedence and associativity rules are summarized here.

4 The following table gives the precedence of the MHDL operators from highest to lowest.

```
OPERATORS FROM HIGHEST TO LOWEST PRECEDENCE

Any operator within a designation
The binding between the expression and designation
      in a physical value
Irrefutable patterns (~), as patterns (@)
primitive
Function application
Structure function creation (<==)
if, let, lambda(\), case [leftward]
case [rightward]
List index (!!), function composition (<>)
Exponentiation operators (**, ^, ^^)
Multiplying operators (*, %, /, ``div``, ``mod``,
      ``rem``, ``quot``)
Adding operators (+, -)
List difference (\\), cons (:-), list concatenation (++)
Relational operators and list membership (/=, <, <=, ==, >,
      >=, ``elem``, ``notElem``)
Boolean and (&&)
Boolean or (||)
Array association (:=)
Right-associating function application ($)
Function types (->)
Type constraints (::)
if, let, lambda (\), [rightwards]
Sequences (..)
Generation schemes (<-)
Grouping (,)
Guards (|)
Case alternatives (->)
Value definitions (=)
Separation (;)
```

Note the notation "[leftward]" and "[rightward]" in the table. This is due to the ambiguity that exists in the MHDL grammar as to the extent of lambda abstractions, conditional expressions, let expressions, and case expressions. This is resolved with a metarule that states that each of these constructs extends as far to the right as possible. Consequently, each of these constructs has two precedences, one to its left, which is the precedence used in the grammar; and one to its right, which is obtained by application

of this metarule.

For the associative operators in MHDL, the following table gives the direction of association.

ASSOCIATIVE OPERATOR	DIRECTION OF ASSOCIATION
Irrefutable patterns (~), as patterns (@)	RIGHT
Function application	LEFT
if, let, lambda(\), case	RIGHT
List index (!!)	LEFT
Function composition (.)	RIGHT
Exponentiation operators (**, ^, ^^)	RIGHT
Multiplication operator (*)	LEFT
Addition/subtraction operators (+, -)	LEFT
Cons (:-), list concatenation (++)	RIGHT
Boolean and (&&)	RIGHT
Boolean or (||)	RIGHT
Right-associating function application ($)	RIGHT
Function types (->)	RIGHT

Example:

EXPRESSION	IS EQUIVALENT TO
`a + b * c`	`a + (b * c)`
`a * let ... in b * c`	`a * (let ... in (b * c))`
`f(a,b,c)`	`((f(a))(b))(c)`
`a :- b :- c`	`a :- (b :- c)`
`a ** b (c)`	`a ** (b (c))`

7.1 Function Applications

The purpose of a function application is to *apply* a function. Applying a function in MHDL is similar to "calling" or "invoking" a function in other languages.

Grammar Rules:

```
function_application ::=
    simple_expression ( argument  { , argument } )

argument ::=
    [ expression ]
```

2 A function application is an invocation of the function whose value is defined by the simple expression that precedes the parenthesized argument list.

3 In an intuitive sense, the notation that allows the expression representing an argument to be omitted allows a particular argument to "float" rather than be fixed to a value. In a more precise sense, a function application such as `f(e1,,e2)` is equivalent to `\x -> f(e1,x,e2)`, where x is a new identifier that does not occur in either `e1` or `e2`.

4 Function applications with multiple arguments are equivalent to multiple nested single-argument function applications associated to the left. I.e.,

```
f(e1,e2,e3,...,en)
```

5 is semantically equivalent to

```
(...(((f(e1))(e2))(e3))...(en)).
```

7.2 Operator Applications

1 An operator application is either an invocation of a binary operator or a use of prefix negation on a numeric quantity.

Grammar Rules:

```
operator_application ::=
    expression operator expression
  | - expression

operator ::=
    operator_symbol
  | `` FUNCTION_name ``
```

2 The first form of an operator application is an infix invocation of a binary operator to the arguments represented by the surrounding expressions.

3 In the second form, prefix negation is applied to a value of any type in the class `Num`. Intuitively, the semantics of this operation is to negate the sign of the operand. Technically, prefix negation is an application of the function `negate` in the `Num` class. The precedence of prefix negation is equal to that of the infix `-` operator. This is the only prefix operator in MHDL.

4 There exists an equivalence between function applications and operator applications. In particular, a function may be applied using an operator application by surrounding the function identifier in double backquotes. For example, the form `f(e1, e2)` is semantically equivalent to `e1 ``f`` e2`. Similarly an operator symbol may be applied using a function application by surrounding the operator symbol in parenthesis (see 7.10 [Simple Expressions], page 91). For example, the form `e1 op e2` is semantically identical to `(op)(e1, e2)`.

5 See 9.2 [Separators and Symbols], page 107 for the definition of the operator symbol lexical element.

7.3 Lambda Abstractions

1 A lambda abstraction defines an unnamed function value.

Grammar Rules:

```
lambda_abstraction ::=
    \ ( pattern { , pattern } ) -> expression
```

2 The patterns in a lambda abstraction correspond to the arguments of the function. The variables in these patterns are bound to values when the function value is applied (see 6.3 [Patterns], page 72 for more information). No variable may appear more than once in the set of patterns. Any lambda abstraction is equivalent to one with only variables as patterns (sometimes called a "lambda expression") in the sense that

```
\(p1, ..., pn) -> e
```

3 where the pi are patterns and e is an expression is equivalent to

```
\(x1, ..., xn) -> case (x1, ..., xn) of (p1, ..., pn) -> e
```

4 where the xi are new identifiers.

5 As an example that ties back to the immediately preceding sections, `\(x) -> x*x` is a lambda abstraction whose value is a function that squares its single argument. The expression that defines the result of the lambda abstraction (following the `->` symbol) is an operator application. The function value can be applied as in `(\(x) -> x*x)(4)` which yields the result `16`.

6 See 6.3 [Patterns], page 72, for the definition of a simple pattern.

7.4 Let Expressions

1 A let expression includes new definitions whose scope is restricted to a single expression.

Grammar Rules:

```
let_expression ::=
    let nested_definitions in expression

nested_definitions ::=
    '{' nested_definition ; { nested_definition ; } '}'

nested_definition ::=
    name_definition
  | value_definition
```

2 A let expression contains a set of locally scoped definitions. In particular, the scope of these definitions is the expression immediately following *in* as well as the right-hand side of each value definition in the nested definitions. In general, the nested definitions may be mutually recursive. Pattern bindings in the definitions are matched lazily (i.e., treated as irrefutable patterns). Specifically, the form `let p=e1 in e2` is semantically equivalent to `case e1 of ~p -> e2`. See 6.3 [Patterns], page 72 for a discussion of irrefutable patterns.

3 As an example, `let {x = 3; y = 4;} in x*y` is an expression that yields the value `12`.

4 See 6 [Named Values], page 61 for the description of name definitions and value definitions.

5 **Note:**

> The capability provided by the nested definitions in a let expression is similar to that provided by the nested definitions in a where clause (e.g.,

in a guarded alternative of a case expression (see 7.6 [Case Expressions], page 85). The important difference is that the nested definitions in a let expression are scoped over a single expression while those in a where clause are scoped over a sequence of guarded definitions.

7.5 Conditional Expressions

1 A conditional expression gives a value that depends on the result of a boolean-valued expression.

Grammar Rules:

```
conditional_expression ::=
    if expression then
        expression
    else
        expression
```

2 In a conditional expression, the expression immediately following *if* must be of type `Bool`. The remaining two expressions must be of the same type and this common type is the type of the conditional expression. If the `Bool` expression has the value `True`, then the result of the conditional expression is the value of the expression immediately following *then*; if the `Bool` expression has the value `False`, then the result of the conditional expression is the value of the expression immediately following *else*; otherwise, the value of the conditional expression is $\perp$.

7.6 Case Expressions

1 Case expressions are used to select among a variety of possible alternatives in determining an expression value.

Grammar Rules:

```
case_expression ::=
    case expression of '{' alternatives '}'

alternatives ::=
```

```
    alternative ; { alternative ; }

alternative ::=
    general_alternative
  | guarded_alternative

general_alternative ::=
    pattern -> expression [ where nested_definitions ]

guarded_alternative ::=
    pattern guard_pattern [ where nested_definitions ]

guard_pattern ::=
    guard -> expression [ guard_pattern ]

guard ::=
    '|' expression
```

2 A general alternative is equivalent to the guarded alternative form in which the guarded alternative consists of a single guard pattern, the expression in the guard of that guard pattern is `True`, and the expression in the guard pattern is the same as that in the general alternative.

3 The remainder of this description of the case expression will assume the guarded alternative form.

4 The expressions immediately following the `->` symbols in the guard patterns in the case expression are called the *bodies* of the case expression. The bodies of the case expression must all have the same type and this common type is the type of the case expression.

5 The evaluation of a case expression proceeds by pattern matching the expression immediately following *case* against the alternatives, one alternative at a time, in the order in which they appear. The event of a successful match causes the corresponding body to be evaluated. Under this circumstance it is the result of evaluating the body that defines the value of the case expression. The body is evaluated in the context of the corresponding *where* nested definitions and the bindings created during the match. If there is no successful match, the result of the case expression is $\perp$.

6 The nested definitions, if present, are local in scope to the alternative in which they appear. As with nested definitions in let expressions, the nested definitions in an alternative may be mutually recursive and pattern bindings in these nested definitions are matched lazily (see 7.4 [Let Expressions], page 84).

7 As an example of a case expression

```
case w of {
   [] -> [];
   x:-xs -> f(x, xs)
}
```

8 results in the empty list if w is itself the empty list; otherwise, the result is that of an application of `f` passing as arguments the head of `w` and the tail of `w`, respectively.

9 See 6.3 [Patterns], page 72, for the definition of a general pattern and pattern-matching semantics. See 7.4 [Let Expressions], page 84, for the definition of nested definitions.

10 **Note:**

> A conditional expression is always equivalent to a case expression in that
>
> `if e1 then e2 else e3`
>
> is defined to be the same as
>
> `case e1 of {True -> e2; False -> e3}.`

7.7 Typed Expressions

1 A typed expression is used to explicitly specify the type of an expression.

Grammar Rules:

```
typed_expression ::=
    ( expression :: type_specification [ where context ] )
```

2 The value of a typed expression is that of the expression immediately preceding the `::` symbol. The type indicated by the type specification must be a specialization (see 4 [Types and Classes], page 29) of the principal type of the expression in the typed expression. The principal type of the typed expression is that indicated in the type specification.

3 **Note:**

> Explicitly stating the type of a value with a typed expression is occasionally useful in disambiguating overloaded expressions that would otherwise be incorrect due to the existence of multiple potential meanings.

7.8 Structure Function Creations

1 A structure function creation constructs a *structure function*, a function which encapsulates the behavior of a structure.

Grammar Rules:

```
structure_function_creation ::=
    property_name_list <== STRUCTURE_name property_name_list

property_name_list ::=
    ( PROPERTY_name { , PROPERTY_name } )
```

2 A structure function creation is a form of expression that takes a structure (see 3 [Structure], page 17) and constructs a function out of it in the following fashion. The first property name list is the *output list*, and the second property name list is the *input list*. The notation `tin` is used to indicate the type of the nth property in the input list, and the notation `ton` is used to indicate the type of the nth property in the output list.

3 The principal type of the created structure function is:

```
ti1 -> ... -> tin -> (Bool, to1, ..., tom)
```

4 where there are `n` property names in the input list, and `m` property names in the output list. The value that results from a complete application of this function is defined in an operational sense by the following sequence of steps:

1. A new part for the structure is created. See 3.4 [Elaboration], page 26, for a description of this process.
2. Each argument to the function is paired, in order, with the property name that occurs in the same position in the input list as the argument in the application list. For each pairing, an assignment of the form "`PROPERTY_name = argument`" is added to the newly created part. Each property name in the input list refers to the corresponding named value in the newly created part (not the part in which the structure function occurs).
3. The value of the function application is a tuple. The first element of the tuple is always of type `Bool`. This element is true if and only if all constraints along with any implementation defined checks concerning the consistency of the design in the design hierarchy whose root is the newly created part evaluate to `True`.

Other elements of the tuple are created by reading the properties indicated by the names in the output list, in order, from the newly created part, and placing those values in the corresponding position of the tuple.

7.9 Derivatives and Integrals

1 A derivative in MHDL is the mechanism for taking the derivative of a function in the sense of this term in mathematics. Similarly, an integral is the MHDL language mechanism for taking the integral of a function.

Grammar Rules:

```
derivative_or_integral ::=
    derivative
  | integral

derivative ::=
    derivative ( identifier ) ( FUNCTION_expression )

integral ::=
    integral ( identifier )
        [ ( REAL_expression , REAL_expression ) ]
        ( FUNCTION_expression )
```

2 A *derivative* is a function that when applied to a function gives the mathematical derivative of that function. Similarly, an *integral* gives the mathematical integral of the function. An integral which has an expression pair following the *integral* keyword and the identifier is a *definite integral*; an integral without the expression pair is an *indefinite integral*.

3 The effect of a derivative or integral is defined in terms of an equivalent function application, where the function being applied is one of `deriv`, `defint`, or `indefint` (see 10.5.5 [Numeric Functions], page 176) and the argument passed is an implicit lambda abstraction. Specifically, if `i` is an identifier and `e`, `e1`, `e2` are expressions, the derivative

```
derivative (i) (e)
```

4 is equivalent to

```
deriv (\(i) -> e)
```

5 and the indefinite and definite integrals

```
integral (i) (e)
integral (i) (e1, e2) (e)
```

6 are equivalent to

```
indefint (\(i) -> e)
defint ((\(i) -> e), e1, e2)
```

7 respectively. Thus, as examples,

```
derivative (t) (f(t))
integral (var) (1,0, 2.0) (g(var, 0.1))
integral (x) (x*x + y)
```

8 are equivalent to:

```
deriv (\(t) -> f(t))
defint ((\(var)-> g(var, 0.1)), 1.0, 2.0)
indefint (\(x) -> x*x + y)
```

9 respectively.

10 **Note:**

A consequence of the above rules is that in the derivative `derivative (i) (e)` if the identifier `i` is not referenced in the expression `e`, the function passed to `deriv` is a constant function with respect to `i` (in which case the result of the derivative is a function giving zero for all `i`). While the language does not define this situation to be an error, an implementation is encouraged to issue a warning for a derivative in which `i` is not referenced in `e`. Similar remarks apply for integrals.

11 **Note:**

An expression that uses special names is an implicit lambda abstraction; thus, such expressions can be used as function expressions in derivatives and integrals. See 8.5 [Special Names], page 103, for more details.

7.10 Simple Expressions

1 Simple expressions are the basic building blocks of MHDL expressions.

Grammar Rules:

```
simple_expression ::=
    NAMED_VALUE_name
  | literal
  | null_expression
  | parenthesized_expression
  | tuple
  | list
  | arithmetic_sequence
  | generator
  | physical_value
  | primitive_value
```

2 A simple expression may be the name of a named value or a part of the design hierarchy that corresponds to a definition designator. See 6 [Named Values], page 61 for the description of named values and the means by which they are bound to values. See 8 [Names], page 99 for the description of names.

3 The following subsections detail the remaining syntactic possibilities for simple expressions.

7.10.1 Literals

1 MHDL supports four basic kinds of literals.

Grammar Rules:

```
literal ::=
    integer_literal
  | floating_point_literal
  | character_literal
  | string_literal
```

2 Integer, float, character, and string literals have principal types `Int`, `Float`, `Char`, and `String`, respectively. See 9.5 [Literals], page 110 for the definition of the integer, float,

character, and string lexical elements.

3 **Note:**

> In practice, integer and float literals may be used freely within `Num` and `Fractional` expressions with the necessary type conversions taking place implicitly due to type coercion functions (see 4.3 [Type Coercions], page 51).

7.10.2 Null Expressions

1 Null expressions provide the single value of the null type.

Grammar Rules:

```
null_expression ::=
    ()
```

2 See 4.1.1 [Specified Types], page 30 for a definition of the MHDL null type.

7.10.3 Parenthesized Expressions

1 A subexpression may be grouped together using a parenthesized expression.

Grammar Rules:

```
parenthesized_expression ::=
    ( expression )
```

2 A parenthesized expression is equivalent to the expression contained within parentheses. Parenthesized expressions are used to control or document evaluation order within an expression.

7.10.4 Tuples

1 Tuples are values of a tuple type.

Grammar Rules:

```
tuple ::=
    ( expression , expression { , expression } )
```

2 If the types of the expressions within a tuple are `t1`, `t2`, ..., `tn`, respectively, then the tuple is of the tuple type `(t1, t2,..., tn)`. See 4.1.1 [Specified Types], page 30 for a description of tuple types.

7.10.5 Lists

1 Lists are values of a list type.

Grammar Rules:

```
list ::=
    '[' [ expression { , expression } ] ']'
```

2 The basic constructors for lists in MHDL are `:-` and `[]` (or Nil) as defined in the MHDL package Standard (see 10.3.2 [List], page 149). Intuitively, `:-` prepends a new element to the front of a list and `[]` represents the empty list. The general form of a list:

3 `[e1, e2, ..., en]`

4 is equivalent notation for the list built with these constructors as follows:

5 `e1 :- (e2 :- ( ... (en :- []) ... )).`

6 See 4.2.4 [Lists], page 41 for a description of list types.

7.10.6 Arithmetic Sequences

1 Arithmetic sequences are a convenience in MHDL for defining lists of numeric and character data.

Grammar Rules:

```
arithmetic_sequence ::=
    '[' expression [ , expression ] .. [ expression ] ']'
```

2 The expressions appearing in an arithmetic sequence must all be of the same type and this type must be in the class `Enum` defined in the MHDL package Standard.

3 The formal definition of arithmetic sequences for a particular type is given with the methods `enumFrom`, `enumFromThen`, `enumFromTo`, and `enumFromThenTo` in the type instance of `Enum`.

4 There are four forms of arithmetic sequences and the semantics associated with each form is described in the following paragraphs.

5 An arithmetic sequence of the form `[e1 ..]` is a list beginning with `e1` and continuing (possibly indefinitely) in single increments. For example, `[1 ..]` is equal to `[1, 2, 3, ...]`. Formally, `[e1 ..]` equals `enumFrom(e1)`.

6 An arithmetic sequence of the form `[e1, e2 ..]` is a list beginning with `e1` and `e2` and then continuing this progression (possibly indefinitely). For example, `[0, 2 ..]` is equal to `[0, 2, 4, ...]` and `[0, 0 ..]` is equal to an infinite list of all zeros. Formally, `[e1, e2 ..]` equals `enumFromThen(e1, e2)`.

7 An arithmetic sequence of the form `[e1 .. e3]` is a list beginning with `e1`, continuing with single increments, and terminating with the last value that is less than or equal to `e3`. For example, `[0 .. 3]` is equal to `[0, 1, 2, 3]` and `[3 .. 0]` is equal to an empty list. Formally, `[e1 .. e2]` equals `enumFromTo(e1, e3)`.

8 Finally, an arithmetic sequence of the form `[e1, e2 .. e3]` is a list similar to `[e1, e2 ..]` but where `e3` acts as an upper limit. For example, `[0, 2 .. 7]` is equal to `[0, 2, 4, 6]`. Formally, `[e1, e2 .. e3]` equals `enumFromThenTo(e1, e2, e3)`.

7.10.7 Generators

1 A generator is a powerful and convenient way to generate general data structures whose types are members of the class `Monad`.

Grammar Rules:

```
generator ::=
    '[' expression '|' qualifier { , qualifier } ']'

qualifier  ::=
    pattern <- expression
  | expression
```

2 A qualifier of the form `p <- e` where `p` is a pattern and `e` is an expression is called a *generation scheme*. If the pattern in a generation scheme is of type `a`, the expression

must be of type `M (a)`, where `M` is a type that is an instance of class `Monad`. A qualifier that is in the form of an expression is called a *guard*. In this case, the expression must be of type `Bool`. The results of the generator are as given in the following table:

```
[ e ]                 = result (e)
[ e | p <- exp, qs ] = exp ``bind`` \p -> [ e | qs ]
[ e | exp, qs ]      = if exp then [ e | qs ] else zero
```

In other words, the result of a generator is a structure consisting of a sequence of evaluations of the expression immediately following the left bracket in a sequence of corresponding contexts (i.e., sets of variable bindings). These contexts are obtained by repeatedly matching the pattern in the qualifier against the elements of the list expression and "filtering out" resulting values with the guards. This process takes place one qualifier at a time with variables bound by latter qualifiers "varying faster" than those bound by earlier qualifiers.

A few examples using lists may make these ideas clearer to the reader.

The generator `[z*z | z<-[0..3]]` is equivalent to `[0, 1, 4, 9]`.

The generator `[z*z | z<-[0..3], z/=2]` is equivalent to `[0, 1, 9]`.

Finally, the generator `[(a,b) | a<-[1..3], b<-[1..a]]` is equivalent to `[(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]`.

See 6.3 [Patterns], page 72, for the definition of a general pattern and a description of the MHDL pattern-matching semantics.

7.10.8 Physical Values

A physical value in MHDL is a measurement of some physical property.

Example:

```
3.0 'ns'                      -- a time value measured in
                              -- nanoseconds
(1, 1, 1) 'Amperes/Meters^2'  -- a volume current density
```

Grammar Rules:

```
physical_value ::=
    REAL_AND_SCALE_expression UNIT_designation
```

2 A physical value is an expression followed by a unit designation. The unit designation identifies the property being measured; for example, a unit designation of `seconds` identifies a time measurement. The type of the expression in the physical value must belong to classes `Real` and `Scale`; that type may not be `Phys`, or any dimension type. The type of the physical value itself is `Phys (a)`, where the polymorphic name `a` corresponds to the type of the expression to which the unit designation is attached.

3 If the unit designation consists of a single unit identifier, then it may be any unit identifier defined in a unit definition. If the unit designation consists of an expression of more than one identifier, then it is subject to the same rules as a unit designation in a unit definition; i.e., no unit identifier in the designation may have a unit conversion in its unit definition.

4 Although a unit designation is not a function call, the expression preceding the unit designation is subject to type coercion just as if it were an argument to a function. See 4.3 [Type Coercions], page 51, for more details on type coercion.

5 See 4.2.12 [Physical Types], page 43, for a more detailed explanation of physical types; see 4.2.12.2 [Unit Definition], page 47, for the syntax of a unit designation. See 7 [Expressions], page 79, for information on the precedence of the association of a unit designation with an expression.

6 **Note:**

> Units can be attached to an arbitrary expression to form a physical value. The association precedence of a unit designation is such that the expression
>
> ```
> tm + 3.0 'secs'
> ```
>
> is equivalent to the expression
>
> ```
> tm + (3.0 'secs')
> ```

7.10.9 Primitive Values

1 A primitive value is one that is defined outside of MHDL.

Example:

```
InttoFloat:: Int -> Float;
InttoFloat = primitive "Integer_to_Real";
```

Grammar Rules:

```
primitive_value ::=
    primitive STRING_expression
```

2 A primitive value is one whose value is defined outside of MHDL. There are many roles for primitive values; for example, a function written in C or other computer languages, a tool which is invoked by the function name, or a communications path to a data base server. A primitive value consists of the keyword *primitive*, followed by an expression of type `String`. The value that results from the expression is used to indicate the tool or external function that is associated with the primitive value in an implementation dependent manner. The principal type of a primitive value is the polymorphic type `a`.

3 Each implementation must provide a form of the primitive value that binds the value to a file name. Reading this value corresponds to reading the entire file. Assigning a value to a named value one of whose other value definitions is a primitive value corresponds to writing the file. The form, and exact semantics, of this file binding is implementation defined.

4 The arguments to a function that is defined by a primitive value are strict; that is, all arguments are evaluated before the primitive argument is invoked.

Chapter 8

1 Names

2 A name is a means of identifying a value or an entity in MHDL.

Grammar Rules:

```
name ::=
    simple_name
  | selected_name

simple_name ::=
    identifier
  | ( operator )

selected_name ::=
    { identifier . } identifier . simple_name
```

3 Definitions and properties are identified by *names*. Normally, names are combinations of identifiers. An operator enclosed in parentheses is a name as well.

4 There are several kinds of names:

Simple names. These names consist of an identifier only, and may serve either as a qualified name or a hierarchical name, as described below.

Qualified names. These names identify definitions that are not duplicated as part of a design hierarchy, but are used to define the values in an MHDL description.

These include names for types, classes, named values, models, packages, and libraries.

Hierarchical names. These names identify a definition or property in the design hierarchy.

Special names. These names have a special meaning in engineering usage, and are treated specially in expressions.

5 A simple name may be either a qualified name or a hierarchical name. A selected name may be a qualified name, a hierarchical name, or a record name. Each of these is covered below.

8.1 Simple Names

1 A simple name is an identifier that indicates a named value or definition that occurs in the same model or package as the simple name. If the simple name occurs within a component or structure, then it may indicate a property, connector or definition within the definition list of the structure or component within which the name occurs, or in the model in which the structure occurs. The named value or definition that a simple name indicates must be unique; indeed, in any context, a simple name must have a unique meaning. Other than within a let expression or a where clause, it is an error to define a simple name with a definition if that simple name is already visible at the site of the definition.

2 **Note:**

> Note that there may be more than one definition for the same simple name. In this case, context, either static or dynamic, is used to indicate which definition is indicated by the simple name. This occurs in cases where a name has been hidden (see 8.6 [Visibility], page 104), in which case the visible definition is chosen, or where a value definition occurs in a class instance (see 4.1.4 [Class Instances], page 39), in which case the types of the arguments to the function are used.

8.2 Record Names

1 A *record name* is a selected name which specifies part of a value which is of a record type (see 4.1.2.1 [Record Types], page 36, for a discussion of record types). The final

simple name of the record name must be a field name, and the name preceding the final simple name must specify a value of a record type. It is an error if the constructor name associated with the value specified by the name does not have the field name in its constructor list. The record name identifies that value which is identified by the field name in the constructor list, i.e. which, in the list of values that make up the entire record value, occurs in the same position as the field name in the constructor list of the specified type definition for the record type.

8.3 Qualified Names

1 A *qualified name* indicates a model or package, or a type, named value, or class defined within a model or package. A qualified name has (potentially) three components, which are simple names separated by dots:

- A library name. If present, the library name must occur in the library list of the model or package in which the qualified name occurs.
- A model or package name. This must be present in any qualified name. The model or package which is indicated by the name is that in whose header the simple name occurs and that is in the indicated library. In general, the model or package thus indicated must exist (have been analyzed) when the model or package containing the qualified name is created (is analyzed); however, this is not the case for names which are resolved during elaboration. Specifically, the model, package, or configuration name used in a component definition, and configuration and structure names used in a configuration clause may not exist when the description unit containing the names have been analyzed. See 3.1.1 [Components], page 19 for a discussion on component definitions, and see 3.2 [Configurations], page 22, for a discussion on configuration clauses. In addition, the model name may always be the name of the model in which the qualified name occurs; if this happens with no library name in the model name, then the model in which the qualified name occurs is assumed.

  The indicated library is either the library named by the library name if the library name is present, or if the library name is not present, it is the first library in the library list satisfying the search criteria. A library satisfies the search criteria if a model or package exists in the library whose simple name agrees with that in the model or package name. See 2.6 [Libraries], page 15, for more information on the occurrence of libraries in library lists.
- A simple name. If this is not present, the qualified name indicates the model or package which is indicated by the model or package name and library name (if any).

2 **Note:**

> The use of the term "analyzed" in the discussion above should be interpreted as an example; individual implementations may vary with respect to the level of processing performed on a description unit by different tools. In general, some form of existence or creation (respectively) must have been demonstrated for the above paragraph to apply.

3 The occurrence of a qualified name implies that the package or model in the qualified name has appeared in the inclusion list of the model, structure, or component definition in which the qualified name appears. It is an error if this is not the case (with the exception of the use of the model name in which the qualified name appears, as explained above, or package Standard, which may always be used regardless of its presence in the inclusion list).

8.4 Hierarchical Names

1 Properties and entities in a design hierarchy may be specified by a *hierarchical name*. A hierarchical name refers to a structure, component, connector, part, constraint, or label that has been created by the design hierarchy. If a hierarchical name is a simple name, then the simple name must either indicate one of the items above, or be an abstract property name. If the simple name is an abstract property name, then it indicates a property attached to the component, structure, or part in whose definition list the hierarchical name appears.

2 Where a hierarchical name is a selected name, the selected name may be subdivided into three sections, any of which may be empty; of course, all three sections may not be empty. (Note that the use of the selected name means that all the simple names in the selected name, and therefore all sections and parts of sections of the hierarchical names, are separated by dots).

3 The first section is a series of *component indications*. A component indication is a triple of a structure name, a component name, and a model name; the structure name and component name may be omitted. A component indication refers to a part associated with the component indicated by the component indication. Each component indication "steps into" the part referred to by the component indication; following component indications refer to entities within that part, and therefore use visibility map of that part. If no structure name appears in the component indication, the name of the structure in which the component indicated by the preceding component indication is used, or for the first component indication the name of the structure in which the name appears is used (this structure must exist). If a model name appears in the component indication, it refers to the part created from that model that is associated

with the component indicated by the component name in the component indication. If the first section is missing, and the name occurs in the definition list of a component and the definition name is a connector name, the part associated with the component is assumed; otherwise, the part containing the hierarchical name is assumed.

4 The second section is a definition name. This definition name must be: a label on a connect list, equation, or value definition; a constraint; a connector; a structure; a component; or a model. The definition may consist of at most three identifiers, separated by a dot; if two or three identifiers are used, the first must be a structure, and indicates a definition in the definition list of the named structure; if three identifiers are used, the second must be a component name found in the structure identified by the first name, and indicates a definition in the definition list of the named component. If no definition name is found, then if the first section is also empty, the model, structure, or component whose definition list contains the hierarchical name is assumed; if the first section is non-empty, then the part indicated by the first section is assumed.

5 The third section is one or more abstract property names. A single property name indicates a property attached to the entity identified by the definition name in the second section, or the part indicated by the first section if the second is omitted, or the component, structure or part whose definition list contains the hierarchical name if both the first and second sections are missing. Additional property names indicate properties attached to the property indicated by the immediately preceding property name. If this section is missing, the entity indicated by the definition name is used, or the part indicated by the first section is used if the definition name is missing. All property names in the property name section must be visible both at the site of the hierarchical name and at the site of the name to which the property is attached.

8.5 Special Names

1 Certain names have particular meaning in engineering environments. They specify independent variables in expressions that define signals, or functions from those variables into some physical quantity. These special names, and their types, are:

t. This indicates a time, and is of type Time.

f. This indicates a frequency, and is of type Frequency.

s. This is the Laplace domain variable, and is of type Rectang.

x. This is a position along the x axis of an arbitrary coordinate system, and is always a numeric value of type Length.

y. This is a position along the y axis of an arbitrary coordinate system, and is always a numeric value of type Length.

z. This is a position along the z axis of an arbitrary coordinate system, and is always a numeric value of type Length.

zd. This indicates a number in the z domain, and is of any numeric type.

2 These special names are treated in expressions as follows. If the expression is not preceded by a lambda list then any of these variables are treated as bound, and occur in an implicit lambda list at the front of the entire expression, where the variables in the lambda list are given in the order in which they first occur in the expression. Thus, these special names are bound unless there is a lambda list that explicitly excludes them in front of, or embedded within and preceding all occurrences of the special name within of the expression in which they occur.

3 A special name may not be the target of a value definition, and may not be given a type in a name definition. Similarly, a special name may not be used as a label, a constraint name, a table name, a type name, a class name, a structure name, a component name, a connector name, an abstract property name, a model name, or a package name. In general, the only legal use for a special name is within an expression where it is bound, explicitly or implicitly as given in this section, by a lambda list.

4 **Note:**

> A function definition may use a special name in the pattern on the left hand side of the value definition for the function. This corresponds to a lambda list on the right hand side of the function definition, and the special name is therefore already bound. See 6.2.1 [Function Bindings], page 65, for more information.

8.6 Visibility

1 For any model or package, there is a mapping of simple names to definitions called the *visibility map*. For those definitions that occur within the package or model, the mapping is from the name occurring in the definition to the definition itself. The presence of a model name in a derivation list, or a package name in an inheritance list, includes the definitions from that model or package (called the *included unit*) into the model or package that contains that name (called the *including unit*). In the absence of a visibility item, the visibility map of the including model or package is the union of the two visibility maps; however, names occuring in generate headers, and names created using those names within generate statements, are never included in visibility maps (and thus never referred to except in the textual scope of the generate statement defining them. See 2.3 [Generate Definitions], page 9, for more details.

2 In the presence of a hiding list, the visibility map of the including unit is the union of the visibility map of the including unit with the visibility map of the included unit minus (eliminating) any mappings of names that occur in the hiding list. In the presence of an exposure list, the visibility map of the including unit is the union of the visibility map of the including unit with only those mappings of names in the visibility map of the included unit that occur in the exposure list. See 2.1 [Inclusion Lists], page 7 for a further discussion of this process.

3 Visibility maps are augmented when a configuration is evaluated during elaboration; see 3.2 [Configurations], page 22, for more details on this process. Such an augmentation occurs by taking the union of the visibility map with the visibility map of a model, structure, or component indicated by the configuration. More specifically:

- When a configuration clause with a structure name is evaluated, the visibility map is augmented with the visibility map of the structure indicated by the structure name in the structure configuration.
- When a configuration clause with a component name is evaluated, the visibility map is augmented with the visibility map of the component indicated by the component simple name.

Chapter 9

1 Lexical Rules

2 An MHDL description is a sequence of lexical elements. The purpose of this section is to define these lexical elements and to describe how they are built up from sequences of individual characters.

3 In this chapter the same BNF-style notation is used in specifying lexical rules that is employed in specifying the syntactic rules for other chapters of this reference manual. However, in this chapter it is understood that white space is not allowed between adjacent grammar symbols unless explicitly stated otherwise.

9.1 Character Set

1 The allowed characters in an MHDL description are partitioned into *graphic characters* and *format effectors*. The graphic characters in MHDL are visually represented by a graphical symbol and map to a unique code of the ISO eight-bit coded character set ISO 8859-1:1987(E) ([12]). The format effectors are the ISO characters called horizontal tabulation, vertical tabulation, carriage return, line feed, and form feed.

9.2 Lexical Elements and Separators

1 Textually, an MHDL description consists of a sequence of lexical elements. A lexical element is either an identifier, a reserved word, a literal (integer, floating-point, character, or string), an operator symbol, a reserved operator, a data table, or a comment.

2 In some situations an explicit separator is required between adjacent lexical elements.

In particular, a separator is required if, in the absence of a separator, lexical interpretation as a single lexical element is possible. In MHDL, a separator is either the space character, a format effector, or the end of a line.

3 In the following section the various kinds of lexical elements are described in detail.

9.3 Identifiers

1 There are two forms of identifiers provided in MHDL: basic identifiers and escaped identifiers.

Grammar Rules:

```
identifier ::=
    basic_identifier
  | escaped_identifier
```

9.3.1 Basic Identifiers

1 A basic identifier is the normal form of identifier in MHDL and consists only of letters, digits, and underscores.

Grammar Rules:

```
basic_identifier ::=
    letter { [ underline ] letter_or_digit }

underline ::=
    _

letter_or_digit ::=
    letter
  | digit

letter ::=
    upper_case_letter
  | lower_case_letter
```

2 An upper case letter is an English upper case letter (i.e., one of "A" through "Z") or one of the non-English upper-case letters from ISO 8859-1 such as "À", "Å", "Æ", "Ñ", "Ö, "Ø", and "Ü", etc.

3 Similarly, a lower case letter is an English lower case letter (i.e., one of "a" through "z") or one of the non-English lower case letters from ISO 8859-1 such as "ß", "ã", "æ", "ê", "ï", "í o", "ø", "ù",etc.

4 A digit is any character in the sequence "0", "1", ..., "9".

5 A basic identifier may not be an MHDL reserved word (see 9.4 [Reserved Words], page 110).

6 **Note:**

> Note that in ISO 8859-1 there are a few lower case letters that do not have upper case equivalents (e.g., "ÿ").

9.3.2 Escaped Identifiers

1 An escaped identifier may consist of any sequence of graphic characters.

Grammar Rules:

```
escaped_identifier ::=
    ` graphic_character { graphic_character } `
```

2 If an escaped identifier is to include the backward tick (i.e., "`") graphic character, it must appear twice consecutively counting as a single backward tick in the escaped identifier.

3 An escaped identifier is never the same as a basic identifier.

4 Examples of identifiers:

```
`MHDL` -- an escaped identifier
`mhdl` -- an escaped identifier different from the above
 mhdl  -- a basic identifier different from all of the above
`$77*` -- another escaped identifier
```

5 **Note:**

Format effectors may not be present in escaped identifiers.

9.4 Reserved Words

The following words are reserved by the language and may not be used as identifiers in MHDL:

```
associate       as              attribute
case            class           coercion
columns         components      component
configuration   connectors      connect
constraint      conversion      default
definition      definitions     derivative
deriving        dimension       each
else            end             equation
except          for             from
function        generate        if
instance        integral        in
includes        let             libraries
member          model           multiplier
names           property        of
package         primitive       report
row             signal          structure
table           then            title
type            unit            use
variations      where           with
connection      set
```

In addition to the words given above, the upper case version of those words formed by taking the word and making the letters all upper case (such as `ASSOCIATE`) is also reserved.

9.5 Literals

There are four kinds of literals in MHDL: integer, floating-point, character, and string.

9.5.1 Integer and Floating-Point Literals

1 Numeric literals in MHDL are either integer literals or floating-point literals.

Grammar Rules:

```
integer_literal ::=
    decimal_integer_part [ exponent ]
  | based_integer_part [ exponent ]

floating_point_literal ::=
    decimal_floating_point_part [ exponent ]
  | based_floating_point_part [ exponent ]

decimal_integer_part ::=
    digits

based_integer_part ::=
    base # extended_digits #

decimal_floating_point_part ::=
    digits . digits

based_floating_point_part ::=
    base # extended_digits . extended_digits #

digits ::=
    digit { digit }

extended_digits ::=
    extended_digit { extended_digit }

extended_digit ::=
    digit
  | letter

exponent ::=
    any_e [ any_sign ] digits

any_e ::=
    E
```

```
    | e

any_sign ::=
    +
  | -

base ::=
    digits
```

2 Integer and floating-point literals may either be expressed in conventional decimal form or in *based literal* form by using a based integer part or a based floating-point part.

3 In either a based integer part or based floating-point part, the allowed values for base are `2`, `8`, and `16`. The specified base affects the allowed extended digits that may appear within the literal. Namely, with a base of `2` the allowed extended digits are "`0`" and "`1`"; with a base of "`8`" the allowed extended digits are "`0`" through "`7`"; with a base of `16` the allowed extended digits are "`0`" through "`9`", "`a`" through "`f`", and "`A`" through "`F`" (upper and lower case are equivalent here). The conventional meaning for a based literal is assumed in each of these cases.

4 In all cases, the base and exponent are expressed in decimal notation.

5 An exponent in an integer literal may not have a minus sign.

6 Examples concerning numeric literals:

```
255          -- An integer literal
2#11111111#  -- An integer literal equivalent to that
             --      immediately above
4E12         -- An integer literal
1.0e-10      -- A floating point literal
4095.0       -- A floating point literal
16#f.ff#E+2  -- A floating point literal equivalent
             --      to that immediately above

-- These are NOT legal numeric literals:
--    E+10
--    1.
--    .95
--    1.0e2.0
--    6#555#
```

9.5.2 Character and String Literals

1 Character and string literals represent one or an arbitrary number of characters, respectively.

Grammar Rules:

```
character_literal ::=
    '' single_character ''

string_literal ::=
    " { single_character } "

single_character ::=
    graphic_character
  | escaped_character
  | gap

escaped_character ::=
    \ special_character
  | \ ascii_escape
  | \ digits
  | \ o digits
  | \ x extended_digits

special_character ::=
    a | b | f | n | r | t | v | \ | " | ' | &

ascii_escape ::=
    ^ control_character
  | NUL | SOH | STX | ETX | EOT | ENQ | ACK
  | BEL | BS  | HT  | LF  | VT  | FF  | CR
  | SO  | SI  | DLE | DC1 | DC2 | DC3 | DC4
  | NAK | SYN | ETB | CAN | EM  | SUB | ESC
  | FS  | GS  | RS  | US  | SP  | DEL

control_character ::=
    upper_case_letter
  | @
  | [
  | \
```

```
    | ]
    | ^
    | &

gap ::=
    \ white_space \
```

2 Representing the single forward tick as a character literal requires escaping the forward tick within the character literal using an escaped character, i.e., "\". Similarly, representing a double quote within a string literal requires escaping the double quote, as does the caret character (^). Finally, representing the backslash in either a character literal or string literal requires escaping the backslash.

3 The letters a, b, f, n, r, t, and v as a special character in an escaped character are used as host-independent representations for "alert", "backspace", "form feed", "new line", "carriage return", "horizontal tab", and "vertical tab".

4 Escape characters from the ASCII character set may be represented using an ascii escape. An ascii escape beginning with '^' is the means provided for representing ASCII control characters.

5 An escaped character including digits or extended digits is a numeric escape based on the underlying character representation. Use of numeric escapes is in general non-portable. Decimal, octal, and hexadecimal forms of numeric escapes are possible. For example, `''\10''`, `''\o12''`, and `''\xa''` represent the same character literal.

6 Ambiguities concerning escaped characters in character literals and string literals are resolved using the metarule that says to maximize the character count in the escaped characters. The escape character `\&` has the meaning of "no character" and may be used to inhibit this metarule. For example, `"\46"` is a string literal of length one while `"\4\&6"` is a string literal of length two. Similarly, `"\SOH"` is a string literal of length one and `"\SO\&H"` is a two-character string literal. A `\&` escape character is illegal in a character literal.

7 Finally, a gap is a character sequence consisting of white space enclosed in "\" characters. Gaps do not contribute to the content of string literals. Gaps are typically used for strings that span multiple lines. For example,

```
"abcdefghijklmnopqrst\
   \uvwxyz"
```

8 is a string literal of length 26. A gap is illegal in a character literal. See 9.7 [Data Tables], page 115 for the definition of white space.

9 String literals are a syntactic convenience for expressing lists of characters.

Note:

Format effectors may not be present in string literals except as part of a gap.

9.6 Operator Symbols and Reserved Operators

In general an operator in MHDL is a sequence of characters from the following set

```
! # $ % & * + . / < = > ? @ \ ^ | : _ ~ -
```

Operators fall into two categories: operator symbols and reserved operators. Reserved operators are those used in defining the syntax of MHDL. For example:

```
::      -- used in name definitions
=       -- used in value definitions
\       -- used in lambda abstractions
```

A complete list of these reserved operators follows:

```
->        .          ..        :          ::
<-        <==        ~         =          |
@         \          ^         _
```

Any operator that is not a reserved operator is available for use as an operator symbol. For example:

```
==      -- an operator symbol defined in the package Standard
+       -- an operator symbol defined in the package Standard
#=#     -- an operator symbol available for definition
```

9.7 Data Tables

A data table is a lexical element intended for use in an MHDL table definition.

Grammar Rules:

```
data_table ::=
    '{{' { row } '}}'

row ::=
    table_entry { '|' table_entry } ;

table_entry ::=
    white_space true_table_entry white_space

true_table_entry ::=
    { graphic_character }
```

2 White space within a data table is either an empty sequence of characters, a format effector followed by white space, the blank character followed by white space, or the end of a line followed by white space. A data table has an ambiguous lexical definition and is assembled by the metarule that says to maximize the character count in white space (at the possible expense of the character count in true table entries).

3 A data table can be thought of as a collection of true table entries, where each true table entry has a unique identification in terms of row number and column number. The row number of a true table entry assumes the rows in a data table are numbered sequentially starting from one; the column number of a true table entry assumes the table entries within a row are numbered sequentially starting from one.

4 To represent any of the characters "|", ";", or "\" within a true table entry, the character must be escaped by being preceded immediately by the "\" character. Two consecutive "}" characters may not appear within a true table entry.

5 See 2.4 [Tables], page 13 for the semantics of a data table and its role in an MHDL table definition.

6 **Note:**

Format effectors may not be present in true table entries.

9.8 Comments

1 There are two forms of comments in MHDL: *ordinary comments* and *nested comments*. Regardless of the form of the comment, there is no semantics associated with a comment; the purpose of a comment is to provide information to the human reader of the MHDL description.

2 Ordinary comments in MHDL begin with two consecutive dashes (i.e., `--`) anywhere in the MHDL text and extend until the end of the line containing the dashes.

3 Nested comments begin with `{-` and terminate with a corresponding `-}`. Within a nested comment, only character sequences `{-` and `-}` are of significance; `{-` begins a new, inner nested comment and `-}` terminates such an inner comment. Nested comments may be nested in this fashion to any depth.

4 Within an ordinary comment, there is no particular significance to character sequences `{-` or `-}` appearing within the ordinary comment. Similarly, within a nested comment, there is no particular significance to the character sequence `--` appearing within the nested comment.

Chapter 10

1 Standard Definitions

2 Certain standard definitions are included with every MHDL system. There are two kinds of standard definitions: standard models, and definitions which are included in every MHDL model. Standard models are assumed to be defined in a library called `MHDL_Standard`. Definitions included in every model and package are defined in a package called `Standard`. All models and packages are treated as if they have `Standard` in their inheritance list. The contents of `MHDL_Standard` and `Standard` are defined below.

3 An MHDL implementation has the obligation to provide all the values and definitions in package `Standard`, and `MHDL_Standard`. The implementation of these values is up to the implementor. Where a value definition appears in the definitions below, it specifies the value and does not imply an obligation to implement the value using the value definition. The implementor is expected to provide implementations that correspond to the laws of mathematics; for example, addition should be commutative.

4 **Note:**

> The word "MHDL" is used as a prefix to definitions in `MHDL_Standard`, and the word "Standard" is used as a suffix for packages in that library. It is therefore a good idea to avoid using these names in user defined models and definitions.

10.1 Numeric Types

1 Numeric types are types which are members of the various classes found below. In general, they correspond to the numeric types found in most programming languages.

2 The basic class `Eq`, found below, is defined on all numeric types. In some cases (for example, with Funcsigs; see 10.4.1 [Universal Signals], page 163), the definition of equality is not definable. As these functions are necessary in equations which specify mathematical relationships, the implementation must provide these functions; however, they are permitted to diverge upon evaluation. It is the responsibility of the implementation to take appropriate action when these functions are invoked in equations, and filter error messages in accordance with user needs. See 6.5 [Equations], page 76, for more details.

10.1.1 Eq

1 Class `Eq` specifies equality and inequality.

```
package Standard

  class Eq (a) {
    (==), (/=)          :: a -> a -> GenBool(a);
    x /= y              = case x == y of {
    GenTrue   -> GenFalse;
    GenFalse  -> GenTrue;
    otherwise -> ForAll(x, y, funNeq); }; };

  instance Eq (a -> b) {
    x == y = ForAll (x, y, funEq);
    x /= y = ForAll (x, y, funNeq);
  } where Eq (b);

  GenBoolFunctoBoolFunc1:: (a -> GenBool(a)) -> a -> Bool;
  GenBoolFunctoBoolFunc1(f,c)  = toBool(f(c));

  GenBoolFunctoBoolFunc2:: (a -> a -> GenBool(a)) -> a -> a -> Bool;
  GenBoolFunctoBoolFunc2(f,a,b)  = toBool(f(a,b));
```

10.1.2 Ord

1 Class `Ord` specifies a total order on elements of any type in this class.

```
class Ord (a)  {
  (<), (<=), (>=), (>):: a -> a -> Bool;

  x <  y              =  x <= y && x /= y;
  x >= y              =  y <= x;
  x >  y              =  y <  x; }
where { Eq(a) };
```

10.1.3 Group

1 We now begin the numeric class definitions. These are split into a specific hierarchy, which is specified by the context clauses on each class. Class `Group` specifies basic numeric definitions, which all numeric types should implement. The normal algebraic laws of groups must apply to any implementation. We do not include a unit definition (i.e., a zero element); such a unit must be definable for each type in the class `Group`, usually as a literal or a coercion from a literal such as 0.

```
class Group (a)  {
  coercion fromInt    :: Int -> a;
  negate              :: a -> a;
  (+), (-)            :: a -> a -> a;}
where Eq (a);
```

10.1.4 Num

1 The class `Num` includes those types that may be arranged on a normal number line. Thus, the sign and absolute value of the number are both defined.

```
class Num (a)  {
  abs, signum         :: a -> a;}
where Group (a);
```

10.1.5 Mult

1 The following classes define multiplication. The class `Mult` defines a general multiplication operator.

```
class Mult (a)  {
  (*)       :: a -> a -> a; };
```

10.1.6 Ix

The class `Ix` contains all types that may be used as an index, particularly into arrays. Each index type may form a range, may be used to specify an index, may be tested for being in range.

```
class Ix (a)   {
  range              :: (a,a) -> [a];
  index              :: (a,a) -> a -> Int;
  inRange            :: (a,a) -> a -> Bool; }
where Ord(a);
```

10.1.7 Enum

Enumerated types are members of class `Enum`. These are types which may be placed in an ordered list; intuitively, they are types which may be iterated over.

```
class Enum (a) {
  ord                :: a -> Int;
  val                :: Int -> a;
  enumFrom           :: a -> [a];            -- [n..]
  enumFromThen       :: a -> a -> [a];       -- [n,n1..]
  enumFromTo         :: a -> a -> [a];       -- [n..m]
  enumFromThenTo     :: a -> a -> a -> [a];  -- [n,n1..m]

  enumFromTo(n,m)     =  takeWhile((>=) (m), enumFrom(n));
  enumFromThenTo(n,n1,m)
                      =  takeWhile(((if n1 >= n then (>=) else (<=))(m)),
                                   (enumFromThen(n,n1)));  }
where Ix(a);
```

10.1.8 Integers

The class `Integers` consists of types to which ordinary integer arithmetic may be applied.

```
class Integers (a)    {
  div, rem, mod      :: a -> a -> a;
  divRem             :: a -> a -> (a,a);
```

```
  even, odd           :: a -> Bool;

  x ``div`` y         =  q  where { (q,r) = divRem (x, y); };
  x ``rem`` y         =  r  where { (q,r) = divRem (x, y); };
  x ``mod`` y         =  if signum (x) == - signum (y) then r + y else r
                         where { r = x ``rem`` y; };
  even(x)             =  x ``rem`` 2 == 0;
  odd                 =  not <> even;
}
where { Mult (a), Num (a) } ;
```

10.1.9 Fractional

1 The class `Fractional` covers those types for which division and reciprocal are defined. These include rational types.

```
class Fractional (a)  {
  coercion fromFloat  :: Float -> a;
  (/)                 :: a -> a -> a;
  recip               :: a -> a;
  recip (m)           = 1 / m;
}
where { Mult (a), Group (a) };
```

10.1.10 Real

1 The `Real` class defines functions on real numbers. They are the standard mathematical functions, and have their standard definition.

```
class Real (a)        {
  pi                  :: a;
  e                   :: a;
  exp, log, ln, sqrt  :: a -> a;
  (**), logBase       :: a -> a -> a;
  sin, cos, tan       :: a -> a;
  asin, acos, atan    :: a -> a;
  sinh, cosh, tanh    :: a -> a;
  asinh, acosh, atanh :: a -> a;
  toRational          :: a -> Rational;
```

```
  ln (x)              =  log(x) / log(e);
  x ** y              =  exp (log(x) * y);
  sqrt(x)             =  x**0.5;
  logBase(x,y)        =  log(y) / log(x);
  tan (x)             =  sin (x) / cos (x);
  tanh(x)             =  sinh(x) / cosh(x); }
where Fractional (a) ;
```

10.1.11 RealFrac

1 The class `RealFrac` contains function definitions that allow the user to manipulate fractional quantities in a type independent way. These include truncation and rounding definitions. The function `properFraction` takes a fractional number `x` and returns a pair comprising `x` as a proper fraction: an integral number with the same sign as `x` and a fraction with the same type and sign as `x`, and with absolute value less than 1.

```
class RealFrac (a,b)    {
  properFraction      :: a -> (b,a);
  truncate, round     :: a -> b;
  ceiling, floor      :: a -> b;

  truncate(x)          = n where { (n, r) = properFraction(x); };
  round(x)             = if sig == -1
                          then n
                          else if sig == 0
                                  then if even(n) then n else m
                                  else m
                           where {sig = signum(abs(r-0.5));
                                  (n, r) = properFraction(x);
                                  m = if r < 0
                                       then n-1
                                       else n+1; };
  ceiling(x)           = if r > 0 then n+1 else n
                         where { (n, r) = properFraction(x); };
  floor(x)             = if r < 0 then n-1 else n
                         where { (n, r) = properFraction(x); };
}
where {Enum(a), Fractional(a), Num(a), Integers(b)};
```

10.1.12 RealFloat

1 The `RealFloat` class contains a number of functions that allows the model writer to manipulate the various portions of a floating point number, including extracting the radix, the digits, the range, and decoding into various parts.

```
class RealFloat (a)  {
  floatRadix          :: a -> Int;
  floatDigits         :: a -> Int;
  floatRange          :: a -> (Int,Int);
  decodeFloat         :: a -> (Int,Int);
  encodeFloat         :: Int -> Int -> a;
  exponent            :: a -> Int;
  significand         :: a -> a;
  scaleFloat          :: Int -> a -> a;

  exponent(x)         =  if m == 0 then 0 else n + floatDigits(x)
                         where { (m,n) = decodeFloat(x); };
  significand(x)      =  encodeFloat(m,(- (floatDigits(x))))
                         where { (m,_) = decodeFloat(x); };
  scaleFloat(k,x)     =  encodeFloat(m,(n+k))
                         where { (m,n) = decodeFloat(x); } ;  }
where { RealFrac (a,Int), Real (a) };
```

10.1.13 Int

1 The `Int` type represents normal integers, as represented on most machines. The instances for this type are as follows.

```
type Int;

instance Eq (Int) {
  (==)                =  primitive "EqInt";
  (/=)                =  primitive "NotEqInt"; } ;

instance Ord (Int) {
  (<)                 =  primitive "LtInt";
  (<=)                =  primitive "LeInt";
  (>=)                =  primitive "NotLtInt";
  (>)                 =  primitive "NotLeInt"; } ;
```

```
instance Group (Int) {
  fromInt(x)          =  x;
  negate              =  primitive "NegInt";
  (+)                 =  primitive "PlusInt";
  (-)                 =  primitive "MinusInt";  } ;

instance Num (Int) {
  abs                 =  primitive "AbsInt";
  signum              =  primitive "SigInt"; } ;

instance Mult (Int){
  (*)                 =  primitive "MulInt";  } ;

instance Integers (Int) {
  divRem                 =  primitive "DivRemInt"; } ;

instance Ix (Int) {
  range((m,n))        =  [m..n];
  index((m,n),i)      =  i - m;
  inRange((m,n),i)    =  m <= i && i <= n; } ;

instance Enum (Int) {
  ord(n)              =  n;
  val(n)              =  n;
  enumFrom(n)         =  enumFromBy(n,1);
  enumFromThen(n,m)   =  enumFromBy(n,(m - n)); } ;
```

10.1.14 Ratio

1 The `Ratio` type is a type which is a rational number of any integral type. The `Rational` type is a type which is a rational number of `Int`.

```
type  Ratio (a)         = a // a where { Integers (a) };
type  Rational          = Ratio (Int);

instance Eq (Ratio (a)) {
  (a//b) == (c//dd) = a == c && b == dd; }
where Integers (a);
```

```
instance Ord (Ratio (a)) {
  (x//y) <= (x1//y1)  =  x * y1 <= x1 * y; }
where { Ord(a), Integers (a) } ;

instance Group (Ratio (a)) {
  fromInt             =  primitive "InttoRatio";
  negate((x//y))      =  (-x) // y;
  (x//y) + (x1//y1)   =  ((x*m) ``div`` y + (x1*m) ``div`` y1) // m
                         where { m = lcm(y,y1); } ;
  (x//y) - (x1//y1)   =  ((x*m) ``div`` y - (x1*m) ``div`` y1) // m
                         where { m = lcm(y,y1); } ; }
where Integers (a);

instance Num (Ratio (a)) {
  abs((x//y))         =  abs(x) // y;
  signum((x//y))      =  signum(x) // 1;
}
where Integers (a);

instance Mult (Ratio (a)) {
  (x//y) * (x1//y1)   =  reduce((x * x1),(y * y1)); }
where Integers (a);

instance Fractional (Ratio (a)) {
  fromFloat           =  primitive "FloattoRatio";
  (x//y) / (x1//y1)   =  (x*y1) // (y*x1); }
where Integers (a) ;

instance RealFrac (Ratio (a),a) {
  properFraction(x//y)=  (q,r // y)
                           where { (q,r) = divRem(x,y); }; }
where { Integers(a)};
```

10.1.15 Float

1 The `Float` type corresponds to the floating point type found in most programming language implementations. Note that type `Float` has no mathematical functions defined on it; this is because all mathematical operations on floating point numbers allow for physical units, and type `Float` is coerced to type `Physical` before such functions are

defined on it. We assume that `Float` corresponds to the IEEE floating point standard; therefore, we provide functions to test the various kinds of non-numeric values.

```
type Float;
isNumeric, isUndefined, isUnknown, isPInfinity, isNInfinity,
isPInfintesimal, isNInfintesimal:: Float -> Bool;
isNumeric       = primitive "isNumeric";
isUndefined     = primitive "isUndefined";
isUnknown       = primitive "isUnknown";
isPInfinity     = primitive "isPInfinity";
isNInfinity     = primitive "isNInfinity";
isPInfintesimal = primitive "isPInfintesimal";
isNInfintesimal = primitive "isNInfintesimal";
```

10.1.16 Complex

1 The class `Complex` defines the class of complex types, which may be of any type of class Real. A complex type consists of a pair of numbers, which represent a location in a complex coordinate system. Two instances of the complex type are provided: a simple pair, which represents rectangular coordinates, and `Polar`, which represents polar coordinates. Rectangular coordinates are regarded as the default; in the case of operations on different types of complex numbers, the rectanguar one is chosen.

2 As with type `Float`, mathematical operations on types `Rectang` and `Polar` are accomplished by coercion to types `Phys(Rectang)` and `Phys(Polar)`, respectively.

```
class Complex (a, b)  {
  mkPolar             :: b -> b -> a;
  mkRectang           :: b -> b -> a;
  realPart            :: a -> b;
  imagPart            :: a -> b;
  cmag                :: a -> b;
  cphase              :: a -> b; } ;

class ComplexFunc (a)  {
  conjugate          :: a -> a; }
where Real (a);

type Rectang  = (Float, Float);

instance Complex (Rectang, Float) {
```

```
  mkPolar             = primitive "MakePolarRectang";
  mkRectang           = primitive "MakeRectangRectang";
  realPart            = primitive "RealPartRectang";
  imagPart            = primitive "ImagPartRectang";
  cmag                = primitive "MagRectang";
  cphase              = primitive "CphaseRectang"; } ;

type Polar  = Polar (Float, Float);

instance Complex (Polar, Float) {
  mkPolar             = primitive "MakePolarPolar";
  mkRectang           = primitive "MakeRectangPolar";
  realPart            = primitive "RealPartPolar";
  imagPart            = primitive "ImagPartPolar";
  cmag                = primitive "MagPolar";
  cphase              = primitive "CphasePolar"; } ;
```

10.1.17 Physical

1 Physical quantities measure a specific physical property. See 4.2.12 [Physical Types], page 43, for an additional discussion of physical type; see 7.10.8 [Physical Values], page 95, for the syntax of physical values.

2 The class `Scale`, and the function `scaled_value` represent the operation of scaling one number by another. For scalar values, this corresponds to multiplication. For vectors, this is different from vector multiplication, and corresponds to multiplying each individual element one by the other.

3 The units for the numeric functions are defined to be in accord with common mathematical practice; `abs` makes no change to the units of its argument. The transcendental functions accept values either of dimension `blank`, dimension `Dimensionless`, or of dimension `solid_angle`, and return a value with units `pure_number`.

4 The `Phys` type is a physical type that has a component of any other member of classes `Real` and `Scale`. The type `Physical` uses the type `Float` to particularize the predefined type `Phys`, for the purposes of any particular model.

5 **Note:**

> The definition below defines a single implementation of most mathematical functions for type `Phys`, and a single primitive function for those

functions. There is no requirement that an implementation use this mechanism for implemeting the functions; in particular, individual instances for differing specializations of `Phys(a)` is perfectly acceptable.

```
type UnitInfo;
type Phys (a) = Phys (a, UnitInfo);

type Physical = Phys (Float);

class Scale (a, b) {
  scaled_value:: a -> b -> a;
  offset_value:: a -> b -> a; } ;

instance Scale (Int, Int) {
  scaled_value = (*);
  offset_value = (+); } ;

instance Scale (Ratio (c), Ratio (c)) {
  scaled_value = (*);
  offset_value = (+); }
where { Mult (Ratio(c)), Group (Ratio(c)) };

instance Scale (Rational, Rational) {
  scaled_value = (*);
  offset_value = (+); } ;

instance Scale (Phys (a), Phys (a))  {
  scaled_value (x, y) = x * y;
  offset_value (x, y) = x + (y * unitof(x)); }
where { Mult(Phys(a)), Group (Phys(a)), Real(Phys(a)) };

instance Eq(Phys (a)) {
  (==) = "PhysEqual"; } ;

instance Ord(Phys(Float)) {
  (<)  = "PhysicalLessThan";
  (<=) = "PhysicalLessEqual";
  (>=) = "PhysicalGreaterEqual";
  (>)  = "PhysicalGreaterThan"; } ;

instance Group (Phys (a)) {
```

```
  fromInt           =  primitive "fromIntPhysical";
  negate            =  primitive "NegatePhysical";
  (+)               =  primitive "PlusPhysical";
  (-)               =  primitive "MinusPhysical";  };

instance Num (Phys (a)) {
  abs                 =  primitive "AbsPhysical";
  signum              =  primitive "SigPhysical"; };

instance Mult (Phys (a)) {
  (*)               =  primitive "MulPhysical"; };

instance Fractional (Phys (a)) {
  fromFloat           =  primitive "FromFloatPhysical";
  (/)                 =  primitive "DivPhysical"; } ;

instance RealFrac (Phys(a), Int) {
  properFraction      = primitive "ProperFractionPhysical"; } ;

instance Real (Phys (a)) {
  pi                  =  primitive "PiPhysical";
  e                   =  primitive "EPhysical";
  **                  =  primitive "PowerPhysical";
  sqrt                =  primitive "SqrtPhysical";
  exp                 =  primitive "ExpPhysical";
  log                 =  primitive "LogPhysical";
  sin                 =  primitive "SinPhysical";
  cos                 =  primitive "CosPhysical";
  asin                =  primitive "AsinPhysical";
  acos                =  primitive "AcosPhysical";
  atan                =  primitive "AtanPhysical";
  sinh                =  primitive "SinhPhysical";
  cosh                =  primitive "CoshPhysical";
  asinh               =  primitive "AsinhPhysical";
  acosh               =  primitive "AcoshPhysical";
  atanh               =  primitive "AtanhPhysical";
  toRational          =
           error ("toRational not supported for physical types");
  } ;

instance RealFloat (Phys(Float)) {
  floatRadix          =  primitive "FloatRadixPhys";
```

```
    floatDigits         =  primitive "FloatDigitsPhys";
    floatRange          =  primitive "FloatRangePhys";
    decodeFloat         =  primitive "DecodeFloatPhys";
    encodeFloat         =  primitive "EncodeFloatPhys"; } ;

  instance ComplexFunc (Phys (a)) {
    conjugate           =  primitive "ConjugatePhysical"; }
  where Complex (a);

  base_units:: Phys(a) -> a;
  base_units (Phys (x, un)) = x;

  mks_units :: Phys(a) -> a where Fractional(a)
  mks_units(p@Phys(x, un)) = base_units(p) * mks_convert(un)
where { mks_convert :: UnitInfo -> Float;
mks_convert = primitive "MKSConvert"; };

  -- unitof returns a single value of the base unit of its argument
  -- i.e., unitof(10.0 'mi') returns 1 'meter'
  unitof :: Phys(a) -> Phys(a) where { Fractional(a) };
  unitof(Phys(x,un)) = Phys(fromFloat(1.0), un);

  -- The following two are treated specially.
   dimension Dimensionless;
   unit pure_number of Dimensionless;
   unit radian of Dimensionless conversion 1.0 pure_number
        variations rad, rads, radians;
   unit degree of Dimensionless conversion
        1.7453293E-2 pure_number
        variations deg, degrees;
   unit steradian of Dimensionless conversion 1.0 pure_number
        variations sr, srs, steradians;


   dimension Length;
   unit meter of Length variations m, metre, meters, metres;
   unit inch of Length conversion 0.0254 meter variations inches;
   unit foot of Length conversion 0.3048 meter variations feet, ft;
   unit yard of Length conversion 0.9144 meter variations yards;
   unit mil of Length conversion 0.0000254 meter variations mils;
   unit micron of Length conversion 10.0E-6 meter variations microns;
```

```
dimension Mass;
unit gram of Mass variations g, gm, gms, grams;
unit pound of Mass conversion 453.5929 gram variations lb, lbs, pounds;
unit ounce of Mass conversion 28.3495 gram variations ounces, oz;

dimension Time;
unit second of Time variations s, sec, seconds, secs;
unit minute of Time conversion 60.0 second variations min, mins, minutes;
unit hour of Time conversion 3600.0 second variations hours, hr, hrs;

dimension Current;
unit ampere of Current variations amp, amperes, amps, A;


dimension Temperature;
unit kelvin of Temperature variations K, kelvins, absolute, degK;
kelvin_to_centigrade :: Physical -> Physical;
kelvin_to_centigrade(x) = x - toPhys(273.15);
centigrade_to_kelvin :: Physical -> Physical;
centigrade_to_kelvin(x) = x + toPhys(273.15);
unit centigrade of Temperature conversion kelvin_to_centigrade
     inverse centigrade_to_kelvin kelvin variations Celsius,
     degree_Celsius, degC;
kelvin_to_Fahrenheit :: Physical -> Physical;
kelvin_to_Fahrenheit(x) =
     toPhys(32.0)+toPhys(9.0)/toPhys(5.0)*(x - toPhys(273.15));
Fahrenheit_to_kelvin :: Physical -> Physical;
Fahrenheit_to_kelvin(x)
     = (x-toPhys(32.0))*toPhys(5.0)/toPhys(9.0)+toPhys(273.15);
unit Fahrenheit of Temperature conversion kelvin_to_Fahrenheit
     inverse Fahrenheit_to_kelvin kelvin variations degF;
kelvin_to_Rankene :: Physical -> Physical;
kelvin_to_Rankene(x) = toPhys(9.0)/toPhys(5.0)*(x - toPhys(273.15));
Rankene_to_kelvin :: Physical -> Physical;
Rankene_to_kelvin(x) = toPhys(5.0)/toPhys(9.0)*x + toPhys(273.15);
unit Rankene of Temperature conversion kelvin_to_Rankene
     inverse Rankene_to_kelvin kelvin variations degR;

dimension Luminosity;
unit candela of Luminosity variations cd, candle, candles;

dimension Substance;
```

```
   unit mole of Substance variations mol, moles;

-- DERIVED DIMENSIONS AND UNITS ( SI, CGS, ENGLISH ... )

   dimension Force = 'Mass * Length / Time^2';
   unit newton of Force = 'kg * m / sec^2' variations N, newtons;
   unit dyne of Force conversion 10E-5 newton variations dyn, dynes;
   unit pound_force of Force conversion 4.44822 newton;

   dimension Energy = 'Force * Length' variations Work;
   unit joule of Energy = 'newton * meter' variations J, joules;
   unit erg of Energy conversion 10E-7 joule variations ergs;
   unit calorie of Energy conversion 4.1840 joule
        variations cal, cals, calories;
   unit electronvolt of Energy conversion 1.6021917*10E-19 joule
        variations eV, electronvolts;

   dimension Power = 'Energy / Time';
   unit watt of Power = 'joule / second' variations W, watts;
   unit horsepower of Power conversion 735.499 watt variations hp, hps;
   watt_to_dBm :: physical -> physical;
   watt_to_dBm(x) = 10 * log (x * 1000);
   dBm_to_watt :: physical -> physical;
   dBm_to_watt(x) = 10E(x/10)/1000;
   unit dBm of Power conversion watt_to_dBm
               inverse dBm_to_watt watt;

   dimension Charge = 'Current * Time';
   unit coulomb of Charge  = 'ampere * sec' variations coulombs, C;
   unit electron_charge of Charge conversion 1.6021917 * 10E-19 coulomb
        variations e;

   dimension Potential = 'Power / Current';
   unit volt of Potential = 'watt / ampere' variations volts;

   dimension Phasor_Potential = 'Potential / Frequency';
   unit phasor_volt = 'volt / hertz' of Phasor_Potential
                      variations phasor_volts;

   dimension Phasor_Current = 'Current / Frequency';
   unit phasor_ampere  of Phasor_Current = 'ampere / hertz'
              variations phasor_amperes phasor_amp phasor_amps;
```

```
dimension Resistance = 'Potential / Current'
          variations Impedance, Reactance;
unit ohm of Resistance = 'volt / ampere' variations ohms;

dimension Capacitance = 'Charge / Potential';
unit farad of Capacitance = 'coulomb / volt' variations F, farads;

dimension Inductance = 'Potential * Time / Current';
unit henry of Inductance = 'volt * second / ampere'
                           variations H, henrys;

dimension Magnetic_flux = 'Potential * Time';
unit weber of Magnetic_flux = 'volt * second' variations Wb, webers;
unit maxwell of Magnetic_flux conversion 10E-8 Wb variations Mx;

dimension Area = 'Length^2';
unit square_meter of Area = 'm^2';

dimension Volume = 'Length^3';
unit litre of Volume = 'dm^3' variations liter, liters, litres, l;
-- alternative is unit litre of Volume conversion 10e-3 m^3 variations ...

dimension Linear_velocity = 'Length / Time' variations Velocity, Speed;
unit meter_per_second of Linear_velocity = 'm / sec';
unit speed_of_light of Linear_velocity
     conversion 2.9979250 * 10E8 meter_per_second
     variations velocity_of_light, c;

dimension Mass_density = 'Mass / Length^3';
unit kilogram_per_cubicmeter of Mass_density = 'kg / m^3';

dimension Linear_acceleration = 'Length / Time^2';
unit meter_per_secondsquared of Linear_acceleration = 'm / sec^2';

dimension Angular_velocity = 'Time^(-1)';
unit radian_per_second of Angular_velocity = 'sec^(-1)';

dimension Angular_acceleration = 'Time^(-2)';
unit radian_per_secondsquared of Angular_acceleration = 'sec^(-2)';

dimension Concentration = 'Substance / Length^3';
```

```
unit mole_per_cubicmeter of Concentration = 'mole / m^3';

dimension Luminance = 'Luminosity / Length^2';
unit candel_per_squaremeter of Luminance = 'candela / m^2';

dimension Pressure = 'Force / Length^2';
unit pascal of Pressure = 'newton / m^2' variations Pa, pascals;
unit standard_atmospheric_pressure of Pressure
     conversion 1.01325*10E5 pascal variations P0;

dimension Magnetic_flux_density = 'Magnetic_flux / Length^2';
unit tesla of Magnetic_flux_density = 'weber / m^2' variations T, teslas;
unit gauss of Magnetic_flux_density conversion 10E-4 tesla;

dimension Admittance = 'Current / Potential'
                       variations Conductance, Susceptance;
unit siemen of Admittance = 'ampere / volt'
                            variations S, siemens, Mho, Mhos;

dimension Luminous_flux = 'Luminosity';
unit lumen of Luminous_flux = 'candela' variations lumens;

dimension Illuminance = 'Luminous_flux / Length^2';
unit lux of Illuminance = 'lumen / m^2' variations luxes, lx;

dimension Frequency = 'Time^(-1)';  --  now Frequency =  / Time is illegal
                                    --  but Time^-1 is legal
unit hertz of Frequency = 'second^(-1)' variations Hz;
                                    -- second^-1 is also legal

dimension Torque = 'Force * Length';
unit newton_metre of Torque = 'newton * meter';

dimension Viscosity = 'Pressure * Time';
unit pascal_second of Viscosity = 'pascal * second';

dimension Surface_tension = 'Force / Length';
unit newton_per_meter of Surface_tension = 'newton / m';

dimension Power_density = 'Power / Length^2';
unit watt_per_squaremeter of Power_density = 'watt / m^2';
```

```
   dimension Energy_density = 'Energy / Length^3';
   unit joule_per_cubicmeter of Energy_density = 'joule / m^3';

   dimension Heat_capacity = 'Energy / Temperature';
   unit joule_per_kelvin of Heat_capacity = 'joule / kelvin';

   dimension Specific_heat_capacity = 'Energy / ( Mass * Temperature )';
   unit joule_per_kilogramkelvin of Specific_heat_capacity =
        'joule / ( kg * kelvin )';

   dimension Thermal_conductivity = 'Power / ( Length * Temperature )';
   unit watt_permeter_perdegree of Thermal_conductivity =
        'watt / ( meter * kelvin )';

   dimension Electric_field = 'Potential / Length'
      variations Electric_field_strength;
   unit volt_per_meter of Electric_field = 'volt / meter';

   dimension Magnetic_field = 'Current / Length'
      variations Magnetic_field_strength;
   unit ampere_per_meter of Magnetic_field = 'ampere / meter';

   dimension Electric_flux_density = 'Charge / Length^2';
   unit coulomb_per_square_meter of Electric_flux_density = 'coulomb / m^2';

   dimension Current_density = 'Current / Length^2';
   unit ampere_per_square_meter of Current_density = 'ampere / m^2';

   dimension Resistivity = 'Resistance * Length';
   unit ohm_meter of Resistivity = 'ohm * meter';

   dimension Permittivity = 'Capacitance / Length';
   unit farad_per_meter of Permittivity = 'farad / m';

   dimension Permeability = 'Inductance / Length';
   unit henry_per_meter of Permeability = 'henry / m';

-- SI UNIT PREFIX MULTIPLIERS

   multiplier yotta = 1.0e24;
   multiplier zetta = 1.0e21;
   multiplier exa = 1.0e18 variations E;
```

```
multiplier peta = 1.0e15 variations P;
multiplier tera = 1.0e12 variations T;
multiplier giga = 1.0e9 variations G;
multiplier mega = 1.0e6 variations M;
multiplier kilo = 1.0e3 variations k;
multiplier hecto = 1.0e2 variations h;
multiplier deca = 1.0e1 variations da;
multiplier deci = 1.0e-1 variations d;
multiplier centi = 1.0e-2 variations c;
multiplier milli = 1.0e-3 variations m;
multiplier micro = 1.0e-6 variations u;
multiplier nano = 1.0e-9 variations n;
multiplier pico = 1.0e-12 variations p;
multiplier femto = 1.0e-15 variations f;
multiplier atto = 1.0e-18 variations a;
multiplier zepto = 1.0e-21;
multiplier yocto = 1.0e-24;
```

10.2 Standard Enumerations

1 Standard enumerations cover a large class of standard types which comprise the scalar non-numeric types. This includes `Bool`, `Char`, and complex types made out of them, such as `String`. Two types which are made to assist in writing interfaces with other systems, `Bin` and `Text`, are also included.

2 In contrast with the use established in the section on numeric types, the standard functions for the standard enumerations will be given here. They are sufficiently connected with the type definitions that this should aid clarity.

10.2.1 Bool

1 The `Bool` type is the normal boolean, or truth value, type. It has two values: `True`, and `False`.

```
type Bool = True | False deriving (Eq, Ord, Ix, Enum, Text, Binary);
type GenBool(a) = GenTrue | GenFalse | ForAll (a, a, BoolFlag)
  deriving (Eq, Ord, Ix, Enum, Text, Binary);
type BoolFlag = funEq | funNeq;
```

```
class Boolfunc (a) {
  (&&), (||)              :: a -> a -> a;
  not                     :: a -> a;
  otherwise               :: a;  } ;

instance Boolfunc (Bool) {
  True  && x              =  x;
  False && x              =  False;
  True  || x              =  True;
  False || x              =  x;

  not(True)               =  False;
  not(False)              =  True;

  otherwise               = True; };

instance Boolfunc (a -> b) {
  (f && g) (x)            =  f(x) && g(x);
  (f || g) (x)            =  f(x) || g(x);
  (not (f)) (x)           =  not (f(x));
  otherwise               =  \a -> otherwise; }
where Boolfunc (b) ;
```

10.2.2 Char

1 The type `Char` is the normal character type. A string is a list of characters. Standard functions on characters and strings appear below. The `Char` type conforms to the standard ISO 8859-1. See 9.1 [Character Set], page 107, for a summary of this standard. The deferred type definition is a placeholder for the actual enumeration type.

```
type Char;

instance Eq (Char) {
  (==)                = primitive "EqChar";
  (/=)                = primitive "NotEqChar"; };

instance Ord (Char) {
  (<)                 =  primitive "LtChar";
  (<=)                =  primitive "LeChar";
  (>=)                =  primitive "NotLtChar";
```

```
  (>)                 =  primitive "NotLeChar"; } ;

instance Ix (Char) {
  range((c1,c2))      = [c1..c2];
  index((c1,c2),ci)   = ord (ci) - ord (c1);
  inRange((c1,c2),ci) = ord (c1) <= i && i <= ord (c2)
    where {i = ord (ci); }; } ;

instance Enum (Char)  {
  ord = primitive "ChartoInt";
  val = primitive "InttoChar";
  enumFrom (c)         = map(val, [ord(c) .. 255]);
  enumFromThen(c1, c2) = map(val, [ord(c1), ord(c2) .. lastChar])
                      where {
                          lastChar :: Int;
                          lastChar = if c2 < c1
                                         then 0
                                         else 255;
                      };
} ;

isAscii, isControl, isPrint, isSpace            :: Char -> Bool;
isUpper, isLower, isAlpha, isDigit, isAlphanum  :: Char -> Bool;

isAscii (c)             =  ord(c) < 128;
isControl(c)            =  c < '' '' || c == ''\DEL'';
isPrint(c)              =  c >= '' '' && c <= ''~'';
isSpace(c)              =  c == '' '' || c == ''\t'' || c == ''\n'' ||
                           c == ''\r'' || c == ''\f'' ||
                           c == ''\v'';
isUpper(c)              =  c >= ''A'' && c <= ''Z'';
isLower(c)              =  c >= ''a'' && c <= ''z'';
isAlpha(c)              =  isUpper(c) || isLower(c);
isDigit(c)              =  c >= ''0'' && c <= ''9'';
isAlphanum(c)           =  isAlpha(c) || isDigit(c);

toUpper, toLower        :: Char -> Char;
toUpper(c)| isLower(c)  =  chr (ord(c) - ord(''a'') + ord(''A''))
          | otherwise   =  c;

toLower(c)| isUpper(c)  =  chr (ord(c) - ord(''A'') + ord(''a''))
          | otherwise   =  c;
```

```
type String = [Char];

lines         :: String -> [String];
lines("")     =  [];
lines(s)      =  l :- (if null (s) then [] else lines (tail(t1)))
                        where {(l, t1) = break((==) (''\n''),s); } ;

words         :: String -> [String];
words(s)              =  case dropWhile(isSpace,s) of {
                      "" -> [];
                      s1 -> w :- words(t1)
              where { (w, t1) = break(isSpace, s1); } ; } ;

unlines               :: [String] -> String;
unlines(ls)           = concat (map((\(l) -> l ++ "\n"),ls));

unwords               :: [String] -> String;
unwords([])           = "";
unwords([w])          = w;
unwords((w:-ws))      = w ++ concat(map((:-) (('' '''),ws));

error :: String -> a;
error = primitive "Abort";
```

10.2.3 Text

1 The `Text` class provides the user with the facility to convert most MHDL standard types into textual form. This may be used in passing information to tools and other interfaces that require information in this form.

2 The read routines take a string and return a list of values and the strings that produced them; the `readPrec` allows the user to specify a maximum number of digits for numeric values. The Show commands take a string and a value and produce a string which is the input string with the value appended to it; the `showPrec` also allows the user to specify a maximum number of digits for numeric values. A numeric value of zero specifies no limit.

```
type  ReadS (a) = String -> [(a,String)];
type  ShowS   = String -> String;
```

```
class Text (a)  {
    readsPrec :: Int -> ReadS (a);
    showsPrec :: Int -> a -> ShowS;
    readList  :: ReadS ([a]);
    showList  :: [a] -> ShowS;

    readList    = readParen(False,
       (\(r) -> [pr | ("[",s) <- [lex(r)], pr <- readl(s)]))
      where {readl(s) = [([],t) | ("]",t) <- [lex(s)]] ++ readl1(s);
             readl1(s) = [(x:-xs,v) | (x,t) <- reads(s),
                        (xs,v) <- readl2 (lex(t))];
             readl2((",",t)) = [z | z <- readl1(t)];
             readl2(("]",t)) = [([],t)];
             readl2(_) = []; };
    showList (xs) = showChar(''['') <> showl(xs)
      where {showl([])    = showChar('']'');
             showl([x])   = shows(x) <> showChar('']'');
             showl (x:-xs) = shows(x) <> showChar('','') <> showl(xs); };
    } ;

reads                 :: ReadS (a) where { Text (a) };
reads(s)              =  readsPrec(0,s);

shows                 :: a -> ShowS where { Text (a) };
shows(s)              =  showsPrec(0,s);

showChar              :: Char -> ShowS;
showChar              = (:-);

showString            :: String -> ShowS;
showString            = (++);

showParen     :: Bool -> ShowS -> ShowS;
showParen(b,p)=  if b then showChar(''('') <> p <> showChar(''))'') else p;

readParen     :: Bool -> ReadS (a) -> ReadS (a);
readParen(b,g)=  if b then mandatory else optional
                 where { optional(r)  = g(r) ++ mandatory(r);
                         mandatory(r) =
                            [(x,u) | ("(",s) <- [lex(r)],
                                     (x,t)   <- optional(s),
                                     (")",u) <- [lex(t)]   ]; } ;
```

```
lex             :: String -> (String,String);
lex("")         = ("","");
lex(''-'':-''>'':-s)= ("->",s);
lex(''-'':-s)     = ("-",s);
lex(r@(c:-s))   =
        if      isSpace(c)      then lex (dropWhile(isSpace,s))
        else if isAlpha(c)      then span(isIdChar,r)
        else if isSingleSym(c)  then ([c],s)
        else if c == '':''      then span(isMultiSymOrC,r)
        else if isMultiSym(c)   then span(isMultiSym,r)
        else if isDigit(c)      then lexNum(r)
        else if c == ''\`''       then
                let {(ch,t) = lexLitChar(s); ''\`'':-u = t; } in
                (''\`'' :- ch ++ "`", u)
        else if c == ''"''        then
                let { (str,t) = lexString(s); } in
                (''"'':-str, t)
        else error("Error in lex (PreludeText): bad character")
  where {
        isIdChar(c)        = isAlphanum(c) || c == ''_'' || c == ''\`'';
        isSingleSym(c)     = c ``in_op`` ",;()[]{}_";
        isMultiSym(c)      = c ``in_op`` "!@#$%&*+-./<=>?\^|~";
        isMultiSymOrC(c)   = c == '':'' || isMultiSym(c);
        lexNum(r) = (ds++f, t) where {(ds,s) = span(isDigit(r));
                                      (f,t)  = lexFracExp(s); } ;
        lexFracExp (''.'':-r) = (''.'':-ds++e, t)
                                where {(ds,s) = lexDigits(r);
                                       (e, t) = lexExp(s); } ;
        lexFracExp(s)       = ("",s);

        lexExp (''e'':-''-'':-r) = ("e-"++ds, s)
               where { (ds,s) = lexDigits(r); } ;
        lexExp (''e'':-r)     = (''e'':-ds, s)
               where { (ds,s) = lexDigits(r); };
        lexExp(s)           = ("",s);

        lexDigits(r@(dd:-_)) | isDigit(dd) = span(isDigit(r));

        lexString (''"'':-s)  = ("\"", s);
        lexString (s)      = (ch++str, u)
                             where{(ch,t)  = lexLitChar(s);
```

```
                                    (str,u) = lexString(t); } ; } ;

  lexLitChar      :: String -> (String,String);
  lexLitChar (''\\'':-s) = (''\\'':-esc, t)
      where { (esc,t) = lexEsc(s);
              lexEsc (c:-s) | c ``in_op`` "abfnrtv\\\"`&" = ([c],s);
              lexEsc (''^'':-c:-s) |
                (isUpper(c) || c ``in_op`` "@[]^_\\") = (['']^'',c], s);
              lexEsc (r@(dd:-s)) | isDigit(dd) = span(isDigit(r));
              lexEsc (''o'':-s) = (''o'':-os, t)
                     where { (os,t) = nonempty (
                                      \(c) -> c >= ''0'' &&
                                                c <= ''7'', s) ; } ;
              lexEsc (''x'':-s) = (''x'':-xs, t)
                     where { (xs,t) = nonempty (
                                      \(c) -> isDigit(c) ||
                                              c >= ''A'' &&
                                              c <= ''F'', s); } ;
              lexEsc (r@(c:-s)) | isSpace(c) = (sp++"\\", u)
                                   where {
                                   (sp,t) = span(isSpace(s));
                                   (''\'':-u) = t; } ;

              lexEsc(s) =
                if taketwo ``in_op`` doubles
                then (taketwo, drop(2,s))
                else if takethree ``in_op`` triples
                then (takethree, drop(3,s))
                else error(
("Error in lex (PreludeText) Illegal literal escape sequence: "
  ++ takethree))
   where { takethree = take(3,s);
           taketwo   = take(2,s);
           triples = ["NUL","SOH","STX","ETX","EOT","ENQ","ACK",
                      "BEL","DLE","DC1","DC2","DC3","DC4","NAK",
                      "SYN","ETB","CAN","SUB","ESC","DEL"];
           doubles = ["BS","HT","LF","VT","FF","CR","SO","SI",
                      "EM","FS","GS","RS","US","SP"];
           nonempty(p,r@(c:-s)) | p(c) = span(p,r); }; };

  lexLitChar (c:-s)    =  ([c],s);
```

```
instance Text ([a]) {
    readsPrec (p) = readList;
    showsPrec (p) = showList; }
where Text (a);

instance Text (Int) {
    readsPrec = primitive "TextReadInt";
    showsPrec = primitive "TextShowInt"; } ;

instance Text (Ratio (a)) {
    readsPrec = primitive "TextReadRatio";
    showsPrec = primitive "TextShowRatio"; }
where { Text (a) };

instance Text (Rational) {
    readsPrec = primitive "TextReadRational";
    showsPrec = primitive "TextShowRational"; } ;

instance Text (Float) {
    readsPrec = primitive "TextReadFloat";
    showsPrec = primitive "TextShowFloat"; } ;

instance Text (Polar (a)) {
    readsPrec = primitive "TextReadPolar";
    showsPrec = primitive "TextShowPolar"; }
where { Text (a) };

instance Text (Phys (a)) {
    readsPrec = primitive "TextReadPhysical";
    showsPrec = primitive "TextShowPhysical"; }
where { Text (a) };

instance Text (Char) {
    readsPrec = primitive "TextReadChar";
    showsPrec = primitive "TextShowChar"; } ;
```

10.2.4 Binary

1 The `Binary` type defines an implementation dependent type which intuitively allows MHDL values to be communicated to and from the surrounding environment in "binary" form. The only requirement of an implementation regarding these types is that

if the same value is read and then written (converted with `showBin` and back with `readBin`) the same value should result.

2 The type `File` is a type which identifies an external file or I/O device. Any value of type `File` is identified with a file in an implementation dependent fashion. The name of the file (which may be an escaped identifier) may be used in this mapping. The readFile command reads the entire contents of the file. The writeFile command writes the entire contents of the file. Files of type `String` correspond to text files.

```
type File (a) = File (a);
type IOResponse = Success | IOError (String);
type Bin;

readFile:: File (a) -> (IOResponse, a);
readFile = primitive "ReadFile";
writeFile:: a -> File (a) -> IOResponse;
writeFile = primitive "WriteFile";

class  Binary  (a) {
    readBin             :: Bin -> (a,Bin);
    showBin             :: a -> Bin -> Bin; } ;

instance Binary (Int) {
  showBin = primitive "ShowBinInt";
  readBin = primitive "ReadBinInt"; } ;

instance Binary (Float) {
  showBin = primitive "ShowBinFloat";
  readBin = primitive "ReadBinFloat"; } ;

instance Binary (Double) {
  showBin = primitive "ShowBinDouble";
  readBin = primitive "ReadBinDouble"; } ;

instance Binary (Char) {
  showBin (b) = (primitive "ShowBinInt") (ord (c), b);
  readBin (b) = (chr (i,b1)) where {
       (i,b1) = ((primitive "ReadBinSmallInt") (b),
               (primitive "MaxChar")); }; } ;

instance Binary ([a]) {
    showBin(l,b) = showBin ((length(l) :: Int), sb1(l,b)) where {
```

```
      sb1([],b) = b;
      sb1((h:-t),b) = showBin(h,(sb1(t,b))); };
    readBin(bin) = rbl(len,bin1) where {
       len :: Int;
       (len,bin1) = readBin(bin);
       rbl(0,b) = ([],b);
       rbl(n,b) = (h:-t,b2) where {
         (h,b1) = readBin(b);
         (t,b2) = rbl((n-1),b1);}; }; }
where Binary (a);

instance Binary  (Array (a, b)) {
    showBin(a) = showBin (bounds(a)) <> showBin (elems(a));
    readBin(bin) = (listArray(b,vs), bin2)
               where {(b,bin1) = readBin(bin);
                      (vs,bin2) = readBin(bin1); }; }
where { Ix (a), Binary (a), Binary (b), Binary ((a,a)), Binary ([b])};

instance Binary ((a,b)) {
  showBin((x,y)) = (showBin(x)) <> (showBin(y));
  readBin(b) = ((x,y),b2) where {
                (x,b1) = readBin(b);
                (y,b2) = readBin(b1); }; }
where { Binary (a), Binary (b) };

instance Binary ((a,b,c)) {
  showBin((x,y,z)) = (showBin(x)) <> (showBin(y)) <> (showBin(z));
  readBin(b) = ((x,y,z),b3) where {
                (x,b1) = readBin(b);
                (y,b2) = readBin(b1);
                (z,b3) = readBin(b2); }; }
 where { Binary (a), Binary (b), Binary (c) };

instance Binary ((a,b,c,e)) {
  showBin((a,b,c,e)) = (showBin(a)) <> (showBin(b)) <>
                      (showBin(c)) <> (showBin(e));
  readBin(b) = ((a1,a2,a3,a4),b4) where {
                (a1,b1) = readBin(b);
                (a2,b2) = readBin(b1);
              (a3,b3) = readBin(b2);
              (a4,b4) = readBin(b3); }; }
where { Binary (a), Binary (b), Binary (c), Binary (e) };
```

10.3 Composite Types

1 A *composite type* is a type which is build up of other types. Record types are composite types which are basic to MHDL. See 4.1.2.1 [Record Types], page 36, for more details concerning record types. The other two composite types are lists and arrays.

10.3.1 Monad

1 A *monad* is an instance of the monad class, or a type with the result and bind operations (at least) defined upon them. These operations must satisfy the following laws:

Left Unit `result (a) ``bind`` k = k(a)`

Right Unit `m ``bind`` result = m`

Association `m ``bind`` (\a -> k(a) ``bind`` \b -> h(b)) =`
`(m ``bind`` \a -> k(a)) ``bind`` \b -> h(b)`

2 It is erroneous if an instance of the class `Monad` defines implementations for these operations such that these laws do not hold.

3 **Note:**

> There is an extensive literature on the use of monads in functional programs, which will not be repeated here. Some good references are [13], [7], and [14].

```
class Functor (f) {
    map :: (a -> b) -> (f (a) -> f (b)) ; } ;

class Monad (m) {
    result      :: a -> m (a) ;
    join        :: m (m (a)) -> m (a) ;
    bind        :: m (a) -> (a -> m (b)) -> m (b) ;

    join (x)     = bind (x, id)  ;
    x ``bind`` f = join (map (f, x)) ; }
  where Functor (m);

class Monad0 (m) {
    zero   :: m (a); }
```

```
  where Monad (m);

class MonadPlus (c) {
    (++) :: c (a) -> c (a) -> c (a) ; }
  where Monad0 (c);

filter           :: (a -> Bool) -> m (a) -> m (a) where Monad0 (m);
filter (p, xs)    = [ x | x<-xs, p (x) ] ;
```

10.3.2 List

1 Lists are groupings of values that may be of any type (but all the same type). A special syntax is used for the constructor `:-` and for the constructor `Nil` below. See 4.2.4 [Lists], page 41, for more details concerning this syntax.

```
  type List (a) = a :- List (a) | Nil ;

  head                    :: [a] -> a;
  head (x:-_)              =  x;
  head([]) = error("Error in function head: head of []");

  tail                    :: [a] -> [a];
  tail (_:-xs)             =  xs;
  tail([]) =  error("Error in function tail: tail of []");

  null                    :: [a] -> Bool;
  null([])                =  True;
  null (_:-_)             =  False;

  instance Functor (List) {
    map (f, [])           = [];
    map (f, x:-xs)        = f(x) :- map (f, xs); } ;

  instance Monad (List) {
    result (x)            = [x];
    []     ``bind`` f     = [];
    (x:-xs) ``bind`` f    = f(x) ++ (xs ``bind`` f); } ;

  instance Monad0 (List) {
    zero                  = [] ; } ;
```

```
instance MonadPlus (List) {
  []      ++ ys         = ys;
  (x:-xs) ++ ys         = x :- (xs ++ ys) ; } ;

length :: [b] -> Int;
length(x) = genericLength(x);

genericLength :: [b] -> a where { Integers (a) };
genericLength(xs) = len(0,xs) where {
        len(n,[])     = n;
        len(n,(x:-xs))= len((n + 1),xs); };

sort :: [a] -> [a] where { Ord (a) };
sort = primitive "GeneralSort" ;

foldr                   :: (a -> b -> b) -> b -> [a] -> b;
foldr(f,z,[])           =  z;
foldr(f,z,(x:-xs))      =  f(x,(foldr(f,z,xs)));

foldl                   :: (a -> b -> a) -> a -> [b] -> a;
foldl(f,z,[])           =  z;
foldl(f,z,(x:-xs))      =  foldl(f,(f(z,x)),xs);

foldr1                  :: (a -> a -> a) -> [a] -> a;
foldr1(f,[x])           =  x;
foldr1(f,(x:-xs))       =  f(x,(foldr1(f,xs)));

foldl1                  :: (a -> a -> a) -> [a] -> a;
foldl1(f,(x:-xs))       =  foldl(f,x,xs);

repeat                  :: a -> [a];
repeat(x)               =  xs where { xs = x:-xs; };

take                    :: a -> [b] -> [b] where Integers (a);
take (n, x) = case x of {
                  []    -> [];
                  x:-xs -> if n == fromInt(0)
                               then []
                               else x :- take(n-fromInt(1), xs); };
```

```
drop                    :: a  -> [b] -> [b] where Integers (a);
drop (n, x) = case x of {
                  []    -> [];
                  _:-xs -> if n == fromInt(0)
                              then xs
                              else drop(n-fromInt(1),xs); };

takeWhile               :: (a -> Bool) -> [a] -> [a];
takeWhile(p,[])         =  [];
takeWhile(p,(x:-xs))
            | p(x)      =  x :- takeWhile(p,xs)
            | otherwise =  [];

dropWhile               :: (a -> Bool) -> [a] -> [a];
dropWhile(p,[])         =  [];
dropWhile(p,xs@(x:-xs1))
            | p(x)      =  dropWhile(p,xs1)
            | otherwise =  xs;

span, break             :: (a -> Bool) -> [a] -> ([a],[a]);
span(p,xs)              =  (takeWhile(p,xs), dropWhile(p,xs));
break(p)                =  span (not <> p);

in_op                   :: a -> [a] -> Bool where { Eq (a) };
x ``in_op`` []          =  False;
x ``in_op`` (y:-ys)     =  x == y || x ``in_op`` ys;

reverse                 :: [a] -> [a];
reverse                 =  foldl((\(xs, x) -> x:-xs),[]);

and, or                 :: [Bool] -> Bool;
and                     =  foldr((&&),True);
or                      =  foldr((||),False);

concat                  :: [[a]] -> [a];
concat                  =  foldr((++),[]);

zip                     :: [a] -> [b] -> [(a,b)];
zip                     =  zipWith (\(a, b) -> (a,b));

zip3                    :: [a] -> [b] -> [c] -> [(a,b,c)];
zip3                    =  zipWith3 (\(a, b, c) -> (a,b,c));
```

```
zipWith                 :: (a->b->c) -> [a]->[b]->[c];
zipWith(z,(a:-aas),(b:-bs))=  z(a,b) :- zipWith(z,aas,bs);
zipWith(_,_,_)          =  [];

zipWith3                :: (a->b->c->e) -> [a]->[b]->[c]->[e];
zipWith3(z,(a:-aas),(b:-bs),(c:-cs))
                        =  z(a,b,c) :- zipWith3(z,aas,bs,cs);
zipWith3(_,_,_,_)       =  [];

last                    :: [a] -> a;
last([x])               =  x;
last((_:-xs))           =  last(xs);
last([]) = error("Error in function last (PreludeList): last of []");

init                    :: [a] -> [a];
init([x])               =  [];
init((x:-xs))           =  x :- init(xs);
init([]) = error("Error in function init (PreludeList): init of []");

(!!)                    :: [b] -> a -> b where {Integers (a), Ord(a) };
x !! n   | n < 0 = error ("Error in function !! (PreludeList): n < 0")
         | otherwise = idx(x,n) where {
   idx([],_) = error("Error in function !! (PreludeList): end of list");
             idx((x:-_),0) = x;
             idx((_:-xs),(n+1)) = idx(xs,n); } ;

scanl                   :: (a -> b -> a) -> a -> [b] -> [a];
scanl(f,q,xs)           =  q :- (case xs of {
                                 []   -> [];
                                 x:-xs -> scanl(f,(f(q,x)),xs); } );

scanl1                  :: (a -> a -> a) -> [a] -> [a];
scanl1(f,(x:-xs))        =  scanl(f,x,xs);

scanr                   :: (a -> b -> b) -> b -> [a] -> [b];
scanr(f,q0,[])          =  [q0];
scanr(f,q0,(x:-xs))     =  f(x,q) :- qs
                           where { qs@(q:-_) = scanr(f,q0,xs); } ;

scanr1                  :: (a -> a -> a) -> [a] -> [a];
scanr1(f, [x])          =  [x];
```

```
scanr1(f, (x:-xs))      =  f(x,q) :- qs
                           where {qs@(q:-_) = scanr1(f,xs); } ;

splitAt                 :: a -> [b] -> ([b],[b]) where Integers (a);
splitAt (n, x) = if n == fromInt(0)
                 then ([],x)
                 else case x of {
                     []    -> ([],[]);
                     y:-ys -> (y:-ys1,ys2)
                          where {(ys1,ys2) = splitAt(n-fromInt(1),ys);}; };


nub                     :: [a] -> [a] where { Eq (a) };
nub([])                 =  [];
nub((x:-xs))            =  x :- nub (filter((/=) (x),xs));

scan                    :: (a -> b -> a) -> a -> [b] -> [a];
scan(f,q,xs)            =  q :- case xs of {
                                 []   -> [];
                                 x:-xs -> scan(f,(f(q,x)),xs); } ;

iterate                 :: (a -> a) -> a -> [a];
iterate(f,x)            =  x :- iterate(f,f(x));

cycle                   :: [a] -> [a];
cycle(xs)               =  xs1 where { xs1 = xs ++ xs1; };

partition :: (a -> Bool) -> [a] -> ([a],[a]);
partition(p,xs) = part(p,xs,([], [])) where {
        part(_,[],res) = res;
        part(p,(x:-xs),(left, right))
               | p(x) = part(p,xs,(x:-left, right))
               | otherwise = part(p,xs,(left, x:-right)) ;} ;

sum, product            :: [a] -> a where { Group (a), Mult (a) };
sum(x)                  =  foldl((+),0,x);
product(x)              =  foldl((*),1,x);

-- sums and products give a list of running sums or products from
-- a list of numbers.  For example,  sums [1,2,3] == [0,1,3,6].
sums, products          :: [a] -> [a] where { Group (a), Mult (a) };
sums(x)                 =  scanl((+),0,x);
```

```
products(x)             =  scanl((*),1,x);

max, min :: a -> a -> a where Ord(a);

max(x,y)| x >= y    =  x
        | y >= x    =  y;

min(x,y)| x <= y    =  x
        | y <= x    =  y;

maximum, minimum        :: [a] -> a where { Ord (a) };
maximum(x)              =  foldl1(max,x);
minimum(x)              =  foldl1(min,x);

any, all                :: (a -> Bool) -> [a] -> Bool;
any(p)                  =  or <> map (p);
all(p)                  =  and <> map (p);

-- elem is the list membership predicate, usually written in infix form,
-- e.g., x ``elem`` xs.  notElem is the negation.
elem, notElem           :: a -> [a] -> Bool where { Eq (a) };
elem(x,y)               =  (any <> (==))(x,y);
notElem(x,y)            =  (all <> (/=))(x,y);

transpose               :: [[a]] -> [[a]];
transpose(x)            =  foldr
                             (\(xs,xss) -> zipWith
                                 ((:-), xs,(xss ++ repeat ([]))),
                             [],x);

zip4            :: [a] -> [b] -> [c] -> [e] -> [(a,b,c,e)];
zip4            =  zipWith4 (\(a,b,c,e) -> (a,b,c,e));

zip5            :: [a] -> [b] -> [c] -> [e] -> [f] -> [(a,b,c,e,f)];
zip5            =  zipWith5 (\(a,b,c,e,f) -> (a,b,c,e,f));

zip6            :: [a] -> [b] -> [c] -> [e] -> [f] -> [g]
                   -> [(a,b,c,e,f,g)];
zip6            =  zipWith6 (\(a,b,c,e,f,g) -> (a,b,c,e,f,g));

zip7            :: [a] -> [b] -> [c] -> [e] -> [f] -> [g] -> [h]
                   -> [(a,b,c,e,f,g,h)];
```

```
zip7            =  zipWith7 (\(a,b,c,e,f,g,h) -> (a,b,c,e,f,g,h));

zipWith4        :: (a->b->c->e->f) -> [a]->[b]->[c]->[e]->[f];
zipWith4(z,(a:-aas),(b:-bs),(c:-cs),(e:-es))
                =  z(a,b,c,e) :- zipWith4(z,aas,bs,cs,es);
zipWith4 (_,_,_,_,_)    =  [];

zipWith5        :: (a->b->c->e->f->g)
                   -> [a]->[b]->[c]->[e]->[f]->[g];
zipWith5(z,(a:-aas),(b:-bs),(c:-cs),(e:-es),(f:-fs))
                =  z(a,b,c,e,f) :- zipWith5(z,aas,bs,cs,es,fs);
zipWith5 (_,_,_,_,_,_)    =  [];

zipWith6        :: (a->b->c->e->f->g->h)
                   -> [a]->[b]->[c]->[e]->[f]->[g]->[h];
zipWith6(z,(a:-aas),(b:-bs),(c:-cs),(e:-es),(f:-fs),(g:-gs))
                =  z(a,b,c,e,f,g) :- zipWith6(z,aas,bs,cs,es,fs,gs);
zipWith6 (_,_,_,_,_,_,_) =  [];

zipWith7        :: (a->b->c->e->f->g->h->i)
                   -> [a]->[b]->[c]->[e]->[f]->[g]->[h]->[i];
zipWith7(z,(a:-aas),(b:-bs),(c:-cs),(e:-es),(f:-fs),(g:-gs),(h:-hs))
           =  z(a,b,c,e,f,g,h) :- zipWith7(z,aas,bs,cs,es,fs,gs,hs);
zipWith7 (_,_,_,_,_,_,_,_) =  [] ;

instance Eq ([a]) {
  [] == [] = GenTrue;
  (a:-b) == (c:-e) = a == c && b == e;
  _ == _ = GenFalse; }
where { Eq (a) };

instance Ord ([a]) {
  [] <= _ = True;
  _ <= [] = False;
  (a:-b) <= (c:-e) = a < c || a == c && b <= e;

  _ < [] = False;
  [] < _ = True;
  (a:-b) < (c:-e) = a < c || a == c && b < e; }
where { Ord (a) };
```

10.3.3 Array

1 An *array* is a function representation that consists of an index type and a mapping from consecutive values of that index type and values of any other type (called the *element type*). The array value is given by specifying these mappings, either by a list or with an association list.

2 The function definitions below describe the semantics of arrays. An implementation is free to provide its own implementation of these functions if desired.

3 The monad `StArray` represents a single threaded array. Arrays whose sole access is via the functions `newarray`, `fetch`, and `assign`, along with the standard monad functions defined above (see 10.3.1 [Monad], page 148, for the definition of these functions) are provably single threaded, and may be implemented as a single copy of the array which is updated in place. Implementors may choose to implement these arrays accordingly.

```
type Assoc (a, b) = a := b;
type Array (a, b) = MkArray ((a,a), a -> b) where Ix (a);


makearray     :: (a,a) -> [Assoc (a, b)] -> Array (a, b) where Ix (a);
bounds        :: Array (a, b) -> (a,a) where Ix (a);
(%%)          :: Array (a, b) -> [Assoc (a, b)] -> Array (a, b)
                 where Ix (a);
accum         :: (b -> c -> b) -> Array (a, b) ->
                           [Assoc (a, c)] -> Array (a, b) where Ix (a);
listArray     :: (a,a) -> [b] -> Array (a, b) where Ix (a);
indices       :: Array (a, b) -> [a] where Ix (a);
elems         :: Array (a, b) -> [b] where Ix (a);
assocs        :: Array (a, b) -> [Assoc (a, b)] where Ix (a);
accumArray    :: (b -> c -> b) -> b -> (a,a) ->
                          [Assoc (a, c)] -> Array (a, b) where Ix (a);
amap          :: (b -> c) -> Array (a, b) -> Array (a, c) where Ix (a);
ixmap         :: (a,a) -> (a -> b) -> Array (b, c)
                           -> Array (a, c) where { Ix (a), Ix (b) };
update        :: a -> b -> Array (a, b) -> Array (a, b) where Ix (a);

instance Function (Array) {
  apply (MkArray (_, f)) = f; } ;

bounds (MkArray (b, _))  = b;

makearray ((l,u),la) = MkArray((l,u),f(lb))
```

```
  where {  f :: [Assoc (a,b)] -> a -> b where Eq(a);
           f((x:=y):-ls,x1) = if x1 == x then y
                              else f(ls,x1);
           f([],_)          = error ("Element not found");
           lb = [ (x:=y) | (x:=y) <- la, if x >= l && x <= u then True
                             else error("Association out of range")];
} ;

indices = range <> bounds;

a %% us   = makearray (bounds (a),
         [i := apply(a, i) | i <- indices (a), notAssoc(i,us)] ++ us)
        where { notAssoc (x,[]) = True;
                notAssoc (x,((y:=_):-ys)) = if x == y then False
                                            else notAssoc(x,ys); } ;

accum (f) = foldl (\(a, (i:=v)) -> a %% [i := f(a(i),v)]);

listArray (b, vs) = makearray (b, zipWith ((:=), range (b), vs));

elems (a)               = [a(i) | i <- indices (a)];

assocs (a)              = [i := a(i) | i <- indices (a)];

accumArray (f, z, b)    = accum (f, (makearray (b,
                                    [i := z | i <- range (b)])));

amap (f, a)             = makearray (b, [i := f (a(i)) | i <- range (b)])
                           where { b = bounds (a); };

ixmap (b, f, a)         = makearray (b,[i:=a(f(i)) | i<-range (b)]);

update (i, val, a)      = makearray (b, [j := ua (j)   | j <- range (b)])
                          where {b = bounds (a);
                                 ua (j) = if i==j then val else a(j);}

type State (s,a) = ST (s -> (a,s));

type StArray (b, c, a) = State (Array (b,c), a)
  where Ix (a);

instance Functor (State(s)) {
```

```
  map (f, (ST st) = ST (\(s) -> let (x, s1) = st(s) in (f(x), s1))); } ;

instance Monad (State (s)) {
  result (x)      = ST (\(s) -> (x,s));
  bind (ST(m), f) = ST (\(s) -> let (x,s1) = m(s);
                                    ST (f1) = f(x)
                                in f1 (s1)) ; } ;

setState:: s -> State (s, ());
setState(x) = ST (\(s) -> ((),x));

refState(x):: State (a,a);
refState = ST (\s -> (s,s));

newarray:: (a,a) -> [b] -> StArray(a, b, ());
newarray (bnds, lst) = ST (\s -> (() , listArray (bnds,lst)));

fetch::b -> StArray (a, b, a);
fetch (i)  =  ST (\(x) -> (x(i), x));

assign:: b -> c -> StArray ( b, c, ());
assign (i, val) = ST (\(x) -> (() , update (i, val, x)));

instance Eq (Array (a, b)) {
    a == a1           =  assocs (a) == assocs (a1); }
where { Ix (a), Eq (b), Eq(Assocs(a,b)) };

instance Ord (Array (a, b)) {
    a <=  a1                  =  assocs (a) <=  assocs (a1); }
where { Ix (a), Ord (b) };

instance Text (Array (a, b)) {
    showsPrec (p, a) = showParen (p > 9,
                       showString ("array ") <>
                       shows (bounds (a)) <> showChar (''  '') <>
                       shows (assocs (a)));

    readsPrec (p) = readParen (p > 9,
         (\(r) -> [(makearray (b, aas), u) | ("array",s) <- [lex (r)],
                                        (b,t)       <- reads (s),
                                        (aas,u)      <- reads (t) ]
                ++
```

```
          [(listArray (b, xs), u) | ("listArray",s) <- [lex (r)],
                                    (b,t)           <- reads (s),
                                    (xs,u)          <- reads (t) ]));}
  where { Ix (a), Text (a), Text (b) };

  instance Binary (Array (a, b)) {
      showBin (a) = showBin (bounds (a)) <> showBin (elems (a));

      readBin (bin) = (listArray (b, vs), bin2)
                 where { (b,bin1)   = readBin (bin);
                         (vs,bin2) = readBin (bin1); }; }
  where { Ix (a), Binary (a), Binary (b) };

  instance Eq (Assoc (a1, a2)) {
    (a1 := a2) == (z1 := z2) = a1==z1 && a2==z2;
    _ == _ = False; }
  where { Eq (a1), Eq (a2) };

  instance Ord (Assoc (a1, a2)) {
    (a1 := a2) <= (z1 := z2) = a1<=z1 || a1==z1 && a2<=z2 ;
    (a1 := a2) <  (z1 := z2) = a1<z1  || a1==z1 && a2<z2; }
  where { Ord (a1), Ord (a2) };

  instance Text (Assoc (a1, a2)) {
    showsPrec (dd, (a1 := a2)) =
      showParen (dd > 5,
                showsPrec (6, a1) <> showString (" := ") <>
                showsPrec (6, a2));

    readsPrec (p)  =  readParen (p > 5,
                      (\(r) -> [(x:=y,u) | (x,s)   <- reads (r),
                                           (":=",t) <- [lex (s)],
                                           (y,u)   <- reads (t)])); }
  where { Text (a1), Text (a2) };
```

4 The following functions have varying requirements on the bounds of the arrays that make up the arguments of the function. It is an error if these constraints are violated.

```
  type Gmatrix (a, b, c) = Array (a, Array (b, c)) where { Ix (a), Ix (b)};
  type Matrix (a) = Gmatrix (Int, Int, a);
  type Vector (a) = Array (Int, a);
```

```
instance Group (Gmatrix (a, b, c))  {
  m + n = makearray (bounds (m),
                [ i := makearray (bounds (m(i)),
                       [ j := m(i,j) + n(i,j) | j <- indices(m(i))])
                | i <- indices (m)]);
  m - n = makearray (bounds (m),
                [ i := makearray (bounds (m(i)),
                       [ j := m(i,j) - n(i,j) | j <- indices(m(i))])
                | i <- indices (m)]);
  negate(m) = makearray (bounds (m),
 [ i := makearray (bounds (apply(m, i)),
[ j := negate(apply(apply(m,i),j))
  | j <- indices(apply(m, i))])
   | i <- indices(m)]);
  fromInt   = error("fromInt for Group(Gmatrix(a,b,c)) not supported");
  }
  where Group (c);

instance Mult (Gmatrix (a, a, c)) {
  m * n = makearray (bounds (m),
                [ i := makearray (bounds (n(i)),
                    [ j := sum ([m(i,k)*n(k,j) | k <- indices(m(i))])
                     | j <- indices(n(i))])
                | i <- indices (m)]); }
where { Mult (c), Group(c), Num(c) };

scaleMatrix:: Gmatrix (a, b, c) -> c -> Gmatrix(a, b, c) where Mult(c);
scaleMatrix (m, s) = makearray (bounds(m),
                    [ i := makearray (bounds (m(i)),
                           j := (s * m(i,j) | j <- indices (m(i))])
                    | i <- indices(m)]);

coercion VectortoMatrix:: Vector (a) -> Matrix (a);
VectortoMatrix (v) = makeArray ((0,0), [0 := v]);

instance Scale (Array (a, b), b) {
  scaled_value (x, y) = makearray (bounds (x),
                           [ i := scaled_value(x(i),y) |
                             i <- indices(x) ]);
  offset_value (x, y) = makearray (bounds (x),
                           [ i := offset_value(x(i),y) |
```

```
                                i <- indices(x) ]); }
  where { Ix (a), Scale (b, b) };

  trans:: Gmatrix(a,b,c) -> Gmatrix(b,a,c) where { Ix (a), Ix (b) };
  trans (m) = makearray (bounds (n),
                   [i := makearray (bounds(m),
                                [ j := m(j,i) | j <- indices(m)])
                 | i <- indices(n) ])
             where { (a,b) = bounds(m); n = m(a); } ;

  conj_trans :: Gmatrix(a,b,c) -> Gmatrix(b,a,c)
                where { Ix (a), Ix (b), Complex (c) };
  conj_trans (m) = makearray (bounds (n),
                        [i := makearray (bounds(m),
                                     [ j := conjugate(m(j,i))
                                     | j <- indices(m)])
                      | i <- indices(n) ])
                  where { (a,b) = bounds(m); n = m(a); } ;
```

10.3.4 Exceptions

1 Normal function processing may uncover different errors. In general, it is the obligation of the function writer to provide appropriate error handling. Using the monad style of function definition, many errors can be ignored in large portions of the function code; such errors are called *exceptions*. The class below is provided to encapsulate exceptions; the `GenExcpt` type is provided as an example.

```
  class Exception (a) {
    normal:: a -> Bool; }
  where Monad (a);

  type GenExcpt (a) = GenNorm (a) | GenException;

  instance Functor (GenExcpt (a)) {
    map ( f, GenNorm (a))  = GenNorm (f (a));
    map ( f, GenException) = GenException; } ;

  instance Monad (GenExcpt (a)) {
    result (x)                    = GenNorm (x);
    join (GenNorm (GenNorm (x)))  = GenNorm (x);
```

```
join (GenNorm (GenException)) = GenException;
join (GenException)           = GenException; } ;
```

10.3.5 Topology

1 The `Topology` type represents a topological specification of a shape, as given in the ISO PDES standard on topology ([1]). A layout in MHDL is given as a value of this type.

2 All the types below are drawn directly from the standard above. The directions are assumed to be in the direction of increasing parameter; thus, boolean flags are omitted. Because there is no accompanying geometry definition, the geometric terms are omitted. Optional terms may be provided as empty entries.

```
attribute layout: Topology;

type Topology = Ver (Vertex) | Edg (Edge) | Pat (Path) | Lp (Loop) |
                Fac (Face) | Sbfac (Subface) | Sh (Shell) |
                CES (ConnectedEdgeSet) | CFS (ConnectedFaceSet) ;

type Vertex = D2 (Float, Float) | D3 (Float, Float, Float);

type Edge = Edge (edge_start (Vertex), edge_end (Vertex));

type Path = Path ([Edge]);

type Loop = VerLoop (VertexLoop) | EdgLoop (EdgeLoop) |
            PolLoop (PolyLoop) ;

type VertexLoop = Vertex;

type EdgeLoop = [ Edge ];

type PolyLoop = [ Vertex ];

type Face = Face (outer_bound (Loop), other_bounds ([Loop]));

type Subface = Subface (outer (Loop), other ([Loop]));

type Shell = VerShell (VertexShell) | WrShell (WireShell) |
             OpShell (OpenShell) | ClShell (ClosedShell) ;
```

```
type VertexShell = VertexLoop;

type WireShell = [ EdgeLoop ] ;

type OpenShell = [ Face ] ;

type ClosedShell = [ Face ] ;

type ConnectedEdgeSet = [ Edge ] ;

type ConnectedFaceSet = [ Face ] ;
```

10.4 Signal Types

10.4.1 Universal Signals

1 The class of *universal signals* is the class of all signals from physical quantities to physical quantities. These may include value of the type `Physical`, values of one of the dimension types (see 4.2.12.1 [Dimension Definition], page 44 for more information), or values of various numeric types or vector types that represent physical quantities to the engineer. Universal signals are all those signals in the class `Usig`. `Usig` is a subclass of class `Real`; thus, all signals inherit numeric functions.

2 `Funcsig` is an instance of the class `Usig`. This instance represents functions defined in MHDL.

```
class Usig (f) {
  delay           :: f(a,b) -> a -> f(a,b) where {Real (a), Real(b)};
  sigEq           :: f(a,b) -> f(a,b) -> Bool
                     where {Real(a), Eq(b)};

  -- The following definitions are temporary until the next version
  -- of package Standard

  sigEq (f, g) = primitive "FunctionEquality";
where {Function (f), Num(f), Mult(f)} ;

type Funcsig (b, c) = b -> c
     where { Real (b), Real (c) };
```

```
instance Eq (Funcsig (a, b)) {
  (==)                =  sigEq; } ;

instance Group (Funcsig (a, b)) {
  (negate (f)) (x)    =  negate (f(x));
  (f + g) (x)         =  f(x) + g(x);
  (f - g) (x)         =  f(x) - g(x);
  (fromInt (f)) (x) =
           error("fromInt for Group(Funcsig (a,b)) not supported");
}
where {Group(b)};

instance Num (Funcsig (a, b)) {
  (negate (f)) (x)  =  negate (f(x));
  (abs (f)) (x)     =  abs (f(x));
  (signum (f)) (x)  =  signum (f (x)); } ;

instance Mult (Funcsig (a, b)) {
  (f * g) (x)         =  f(x) * g(x); } ;

instance Fractional (Funcsig (a, b)) {
  (f / g) (x)         =  f(x) / g(x) ; } ;

instance Real (Funcsig (a, b)) {
  pi (x)                =  pi;
  e  (x)                =  e;
  (exp (f)) (x)         =  exp (f (x));
  (log (f)) (x)         =  log (f (x));
  (sqrt (f)) (x)        =  sqrt (f (x));
  (sin (f)) (x)         =  sin (f (x));
  (cos (f)) (x)         =  cos (f (x));
  (tan (f)) (x)         =  tan (f (x));
  (asin (f)) (x)        =  asin (f (x));
  (acos (f)) (x)        =  acos (f (x));
  (atan (f)) (x)        =  atan (f (x));
  (sinh (f)) (x)        =  sinh (f (x));
  (cosh (f)) (x)        =  cosh (f (x));
  (tanh (f)) (x)        =  tanh (f (x));
  (asinh (f)) (x)       =  asinh (f (x));
  (acosh (f)) (x)       =  acosh (f (x));
  (atanh (f)) (x)       =  atanh (f (x));
```

```
  (toRational(f))       =
              error("toRational for Funcsig(a,b) not supported");
  } ;

instance Usig (Funcsig);

class OrdUsig (f) {
  sigLt           :: f(a,b) -> f(a,b) -> Bool
                     where {Real(a), Ord(b)};
  sigLe           :: f(a,b) -> f(a,b) -> Bool
                     where {Real(a), Ord(b)};
  sigGt           :: f(a,b) -> f(a,b) -> Bool
                     where {Real(a), Ord(b)};
  sigGe           :: f(a,b) -> f(a,b) -> Bool
                     where {Real(a), Ord(b)};

  -- The following definitions are temporary until the next version
  -- of package Standard

  sigLt (f, g) = primitive "SignalLessThan";
  sigLe (f, g) = primitive "SignalLessEqual";
  sigGt (f, g) = primitive "SignalGreater";
  sigGe (f, g) = primitive "SignalGreaterEqual";  }
where Usig(f);

type OrdFuncsig (b, c) = b -> c
     where { Real (b), Real (c), Ord(c) };

instance Usig (OrdFuncsig);

instance Eq (OrdFuncsig (a, b)) {
  (==)                =  sigEq; } ;

instance Ord (OrdFuncsig (a, b)) {
  (<)                 =  sigLt;
  (<=)                =  sigLe;
  (>=)                =  sigGt;
  (>)                 =  sigGe; } ;

instance Group (OrdFuncsig (a, b)) {
  (f + g) (x)         =  f(x) + g(x);
  (f - g) (x)         =  f(x) - g(x); } ;
```

```
instance Num (OrdFuncsig (a, b)) {
  (negate (f)) (x)  =  negate (f(x));
  (abs (f)) (x)     =  abs (f(x));
  (signum (f)) (x)  =  signum (f (x)); } ;

instance Mult (OrdFuncsig (a, b)) {
  (f * g) (x)         =  f(x) * g(x); } ;

instance Fractional (OrdFuncsig (a, b)) {
  (f / g) (x)         =  f(x) / g(x) ; } ;

instance Real (OrdFuncsig (a, b)) {
  pi (x)              =  pi;
  (exp (f)) (x)         =  exp (f (x));
  (log (f)) (x)         =  log (f (x));
  (sqrt (f)) (x)        =  sqrt (f (x));
  (sin (f)) (x)         =  sin (f (x));
  (cos (f)) (x)         =  cos (f (x));
  (tan (f)) (x)         =  tan (f (x));
  (asin (f)) (x)        =  asin (f (x));
  (acos (f)) (x)        =  acos (f (x));
  (atan (f)) (x)        =  atan (f (x));
  (sinh (f)) (x)        =  sinh (f (x));
  (cosh (f)) (x)        =  cosh (f (x));
  (tanh (f)) (x)        =  tanh (f (x));
  (asinh (f)) (x)       =  asinh (f (x));
  (acosh (f)) (x)       =  acosh (f (x));
  (atanh (f)) (x)       =  atanh (f (x)); } ;
```

10.4.2 Continuous Signals

1 Continuous signals are assumed to be continuous over the entire range of the independent variable. The continuous signals thus allow differentiation and other mathematical functions on continuous signals.

```
class Cont (f) {
  Laplace      :: f (Time, Phys(Float)) -> f (Frequency, Phys(Float));

  InvLaplace   :: f (Frequency_dim(Rectang), Phys(Rectang)) ->
```

```
    f (Time, Phys(Float));

    Fourier      :: f (Time, Phys(Float)) -> Time -> Time -> Frequency
    -> f (Frequency, Phys(Rectang));
    InvFourier   :: f (Frequency, Phys(Rectang)) -> Frequency -> Frequency
    -> Time -> f (Time, Phys(Rectang));  }
    where {  Function(f)  };

  instance Cont (Funcsig) {
    Laplace = primitive "FuncsigLaplace";
    InvLaplace = primitive "FuncsigInverseLaplace";
    Fourier = primitive "FuncsigFourier";
    InvFourier = primitive "FuncsigInvFourier"; } ;
```

10.4.3 Discrete Signals

1 Discrete signals are signals which are made up of a series of measurements. Discrete signals are represented as a series of samples. The function `next_samp` gets the next sample in a series, and the function `next_time` gets the independent variable value (which is usually of type time) associated with the next sample in the series. Both of these return a discrete signal with the sample just given removed. The `summation` function returns the sum of samples between two independent values, and the z-transform and inverse z-transform have their usual mathematical meaning.

```
  class Disc (f) {
    next_samp:: f(a, b) -> (b, f(a, b)) where {Real (a), Real (b)};
    next_time:: f(a, b) -> (a, f(a, b)) where {Real (a), Real (b)};
    summation:: f(a, b) -> a -> a -> b where {Real (a), Real (b)};
    z_trans  :: f(Time, b) -> f(Frequency, b) where Real (b);
    inv_ztrans:: f(Frequency, b) -> f(Time, b) where Real (b); }
  where Usig (f);
```

10.4.4 Periodic Signals

1 Periodic signals are signals with a distinct period, or frequency. Periodic signals may be either continuous or discrete. The functions `freq` and `phase` extract the frequency and phase of the signal, respectively. The function `mkPer` makes a periodic signal given the frequency, magnitude, and phase.

```
class Per (f)  {
  freq:: f(a) -> Frequency where Real (a);
  phase:: f(a) -> Dimensionless where Real (a);
  mkPer:: Frequency -> Dimensionless -> a -> f (a) where Real (a); }
where Usig (f);
```

2 The class `ContPer` contains the function `pshift`, which represents a phase shift.

```
class ContPer (f) {
  pshift:: f(a, b) -> Dimensionless -> f (a, b) where {Real (a), Real(b)};}
where { Per (f), Cont (f) } ;
```

3 The class `DiscPer` contains the functions corresponding to the discrete Fourier series.

```
class DiscPer (f) {
  DFT::    f (Time, a) -> f(Frequency, a) where Real (a);
  invDFT:: f (Frequency, a) -> f(Time, a) where Real (a); }
where { Disc (f), Per (f) };
```

10.5 Standard Functions

1 Standard functions are functions which are assumed in any MHDL implementation. We will begin with a definition of the class of functions themselves, then go on to standard functions in many categories. Note that many standard functions that operate on the standard enumerations and composite types are given in their respective sections. See 10.2 [Enumerations], page 138, for more details on functions on the standard enumeration types, and see 10.3 [Composite Types], page 148, for definitions of standard functions on lists and arrays.

2 A great number of standard functions are contained within the class definitions for the various signal definitions. See 10.4 [Signal Types], page 163, for a complete definition of these functions.

10.5.1 Function Class

1 The following class defines functions in MHDL. This class name and the function application have special syntax in MHDL; see 4.2.3 [Function Types], page 41, for more details. The `strict` function converts a function from being lazy to being strict in its arguments; i.e., it evaluates all of its arguments before being invoked.

```
class Function (f) {
  apply:: f(a, b) -> a -> b; } ;


(<>):: (b -> c) -> (a -> b) -> a -> c;
f <> g = \(x) -> f (g (x));

strict:: (a -> b) -> a -> b;
strict = primitive "Strict";

instance Function ((->)) {
  apply = primitive "ApplyMHDLFunction"; } ;
```

10.5.2 Coercion Functions

1 A number of coercion functions for the various standard types are built in. They are used to allow the user to write expressions of related types that will be converted to the appropriate type as necessary.

2 Several of the functions are made available for the use of the user; however, they are not designated as coercion functions. Use of these functions may cause loss of information; for example, the loss of the fractional part of a floating point number in the conversion of a `Float` to an `Int`, and the user is required to invoke these functions explicitly when necessary.

3 **Note:**

> Dimension definitions define standard coercion functions as well. See 4.2.12.1 [Dimension Definition], page 44, for more information on the form of these coercion functions.

```
coercion RatiotoFloat:: Ratio(a) -> Float;
RatiotoFloat = primitive "RatiotoFloat";

coercion RatiotoRational:: Ratio(a) -> Rational;
RatiotoRational = primitive "RatiotoRational";

coercion RationaltoRatio:: Rational -> Ratio(a);
RationaltoRatio = primitive "RationaltoRatio";
```

```
coercion RationaltoFloat:: Rational -> Float;
RationaltoFloat = primitive "RationaltoFloat";

ApproxRationaltoRatio:: Rational -> Ratio(a);
ApproxRationaltoRatio = primitive "ApproxRationaltoRatio";

RectangtoPolar:: Rectang -> Polar;
RectangtoPolar (p) = mkPolar (cmag (p), cphase(p));

coercion PolartoRectang:: Polar -> Rectang;
PolartoRectang (p) = mkRectang (realPart (p), imagPart (p));

coercion PhystoTimeFunction:: Phys (a) -> Time_dim (a) -> Phys (a);
PhystoTimeFunction (p) = \(t) -> p;
```

10.5.3 Misc. Functions

1 This section defines a number of useful functions that do not fit into any other category. This includes a number of generally useful numeric functions.

```
gcd             :: a -> a-> a where Integers (a);
gcd (x,y) = if ( x == fromInt(0) && y == fromInt(0) )
              then error("gcd: gcd (0,0) is undefined")
              else  gcd1 (abs (x), abs (y))
                      where { gcd1 (x, y) = if ( y == fromInt(0) )
                                                then x
                                   else gcd1 (y, (x ``mod`` y)); } ;

lcm             :: a -> a-> a where Integers (a);
lcm (x,y) = if ( x == fromInt(0) || y == fromInt(0) )
              then fromInt(0)
              else  abs ((x ``div`` (gcd (x, y))) * y);

(^*) :: a -> b -> a where { Num(a), Mult(a), Integers(b), Ord(b) };
x ^* n = if ( n == fromInt(0) )
              then fromInt(1)
              else if ( n > fromInt(0) )
                      then f1 (x, n-fromInt(1), x)
                      else error ("Error in ^*: Negative exponent found")
                      where {
```

```
                    f1 :: a->b->a->a where {Num(a), Integers(b), Mult(a)};
                    f1 (x,n,y) = if ( n == fromInt(0) )
                                    then y
                                    else g(x,n)
                                    where {
                g (x, n) | even (n)  = g ((x*x), (n``div``fromInt(2)))
                         | otherwise = f1 (x, (n-fromInt(1)), (x*y)); }; };



chr             :: Int -> Char;
chr             = primitive "primInttoChar";

id              :: a -> a;
id (x)          = x;

(^^)            :: a -> b -> a where { Fractional (a), Integers (b),
                                       Ord (b) };
x ^^ n          =  if n >= 0 then x^*n else 1/x^*(-n);


(%)                     :: a -> a -> Ratio (a) where Integers (a);
numerator, denominator  :: Ratio (a) -> a where Integers (a);

numerator (x//y)        =  x;

denominator (x//y)      =  y;

x % y                   =  reduce ((x * signum (y)), (abs (y)));

reduce (x, y)           =  (x ``div`` z1) // (y ``div`` z1)
                           where { z1 = gcd (x, y); };

absReal (x)  | x >= 0    =  x
             | otherwise =  - x;

signumReal (x) | x == 0    =  0
               | x > 0     =  1
               | otherwise = -1;


rangeSize               :: (a,a) -> Int where Ix (a);
```

```
rangeSize ((l,u))       =  index ((l,u), u) + 1;

enumFromBy:: a -> a -> [a] where { Num (a) };
enumFromBy (n, k)       =  n :- enumFromBy ((n+k), k);

floatProperFraction:: a -> (b,a) where { RealFloat (a), Integers (b) };
floatProperFraction (x) =  if n >= 0
                              then (m * b^*n, 0)
                              else (m1, k / c)
                           where { (m,n)  = decodeFloat (x);
                                   b      = floatRadix (x);
                                   (m1,k) = divRem (m, e);
                                   c      = b^*(-n); } ;

floatApproxRational:: a -> a -> Rational where { Real (a),
                      Ord(a), RealFloat(a) };
floatApproxRational (x, eps) =
    case withinEps of {
        r:-r1:-_ | denominator (r) == denominator (r1) -> r1;
        r:-_                                           -> r;
    }
    where {withinEps = dropWhile((\(r) -> abs (r - x) > eps)
                                 (approximants (p, q)));
          (p,q)     = if n < 0 then (m, b^(-n)) else (m*b^n, 1);
          (m,n)     = decodeFloat (x);
          b         = floatRadix (x); };

minInt, maxInt        :: Int;
minInt                =  primitive "minInt";
maxInt                =  primitive "maxInt";

fst                   :: (a,b) -> a;
fst ((x,y))           =  x;

snd                   :: (a,b) -> b;
snd ((x,y))           =  y;

until                       :: (a -> Bool) -> (a -> a) -> a -> a;
until (p, f, x) | p (x)     =  x
                | otherwise =  until (p, f, f (x));
```

10.5.4 Structural Functions

1 A *structural function* is a function that is intended to operate on some MHDL language structure. The types of these functions have a definition designator in their name definitions. See 4.1.1 [Specified Types], page 30, for more information on these kinds of type definitions.

2 The function `full_name` gives the full name of any definition as a string, where the full name is that unique hierarchical name that identifies the definition from within the root of the design hierarchy. The function `local_name` gives the simple name of the selected name that forms the full name of any definition. The function `attribute_value` gives the value of any attribute whose full name is given as the string argument to the function.

3 The function `equations` takes a model, structure, or a component and a regular expression and produces a list of descriptions of the equations, of type `Equation`. This list consists of descriptions of all the equations defined in the definition list of the named entity whose labels match the regular expression. Where the values may be evaluated by a lazy read, the values are given; otherwise, the full names are substituted. The exact form of the regular expression and the exact definition of the type `Equation` are both implementation dependent, and are not further specified here; the type definition `Equation` below is a stub.

4 The function `exists` tests to see if a property is attached to a given definition. It returns true if a property of that name is attached to the referenced definition, and false otherwise.

```
type Equation = Equation;
equations:: definition -> String -> [ Equation ] ;
equations = primitive "ExtractEquations";

full_name:: definition -> String;
full_name = primitive "FullName";

local_name:: definition -> String;
local_name = primitive "LocalName";

attribute_value:: property (a) -> String -> a;
attribute_value = primitive "AttributeValue";

exists:: definition -> property (a) -> Bool;
exists = primitive "Exists";
```

5 The function `connection_set` produces the connection set of any connector, or the list of connectors connected with the given connector (see 3.1.3 [Connection Lists], page 21, for the definition of connection set). The function `connected_obj` takes a connector and an abstract property. It returns a list of all connectors in the connection set of the given connector to which the named property is attached. The return argument is a list of full connector names and values of the property for that name.

6 The functions `structure_conn` takes a structure name, and produces a list of the names of the connectors in the structure. The function `structure_conn_obj` takes a structure name and an abstract property, and returns a list of pairs, where the first is the name of the connector and the second is the value of the property attached to that connector (either directly or by association); only those connectors with property values are returned. The functions `structure_comp` and `structure_comp_obj` perform similar functions on the components of a structure. If the attribute `conn_number` is defined and attached to one or more of the connectors of the structure, then the connectors appear in the lists returned by these functions in the same order as they are given by the values of `conn_number` attached to the various connectors; it is an error if two of these are the same. If one or more of the connectors do not have the attribute `conn_number` attached to them, they appear at the end of the list in any order; however, this order is not dependent upon the other parameters to the function, and does not vary between function applications.

```
attribute conn_number: Int;

connection_set :: connector -> [String];
connection_set = primitive "ConnectionSet";

connected_obj:: connector -> property (a) -> [(String,a)];
connected_obj = primitive "ConnectedObject";

structure_conn:: structure -> [String];
structure_conn = primitive "StructureConnectors";

structure_conn_obj:: structure -> property (a) -> [(String, a)];
structure_conn_obj = primitive "StructureConnectorObject";

structure_comp:: structure -> [String];
structure_comp = primitive "StructureComponents";

structure_comp_obj:: structure -> property (a) -> [(String, a)];
structure_comp_obj = primitive "StructureComponentObject";
```

7 In order to allow operations on the connectors defined in a structure separately from the

connectors defined by the parents of the structure, the following functions are defined. They work similarly to the correspondingly named functions above. Child connectors refer to the connectors defined in the components. All functions work only on the connectors in a connection set or in the structure that are local to the structure itself, i.e. they do not traverse the connection set of the entire design hierarchy, but only of the local structure (between connectors and connectors of components).

```
child_conn_set :: connector -> [String];
child_conn_set = primitive "ChildConnectorSet";

child_con_obj:: connector -> property (a) -> [(String,a)];
child_con_obj = primitive "ChildConnectorObject";

child_str_conn:: structure -> [String];
child_str_conn = primitive "ChildStrructureConnectors";

child_str_obj:: structure -> property (a) -> [(String, a)];
child_str_obj = primitive "ChildStructureObject";
```

8 The function `extract` extracts a value corresponding to the string in a list of string value pairs.

```
extract                :: String -> [(String,a)] -> a;
extract (s, [])        = error ("Error in Extract: String not found");
extract (s, (n,v):-xs) = if s==n then v else extract (s, xs);
```

9 The function `isType` checks to see whether the value can be coerced to the type which is the second argument.

```
isType:: a -> type -> Bool;
isType = primitive "IsType";
```

10 The type `C_type` specifies the type of a connector; the attribute `ctype` holds the type of the connector for a given connector. These are used in the standard packages: See 10.6 [MHDL Standard], page 177, for more information.

11 The other attributes here are for the use of the connector type packages. See 10.6.2 [Connector Types], page 179, for more details.

12 The type `Dir` is used to indicate the direction of a signal flow model.

```
type C_type = no_ctype | electric | sigflow | phasor;
```

```
attribute ctype: C_type;
attribute v:   Time -> Potential;
attribute i:   Time -> Current;
attribute V:  Frequency -> Phasor_Potential_dim(Rectang);
attribute I:  Frequency -> Phasor_Current_dim(Rectang);

type Dir = in_dir | out_dir | inout_dir | no_dir deriving Eq;
attribute dir: Dir;
```

10.5.5 Numeric Functions

1 The following functions are the standard numeric functions of MHDL. These functions are, in general, defined by the implementation. In those cases where the functions return functions, no implication is made concerning whether a numeric or a symbolic solution mechanism is required.

2 The function `deriv` takes the derivative over the first argument. The function `defint` takes a definite integral of the given function over the range given. The function `indefint` gives the indefinite integral of the given function. If applied to function `h`, `indefint` returns a tuple, the first element of which is the calculated function `g`, and the second element of which is the value selected by the solution algorithm as the arbitrary base value `xo` in the formula `g(x) = defint ((x0,x),h)`.

3 The implementation of these functions over the entire range of numeric types may be difficult. Therefore, the implementation of these functions (and, in particular, the integral functions) on `Float` is optional. The implementation of indefint is also optional, as it cannot be implemented by purely numerical methods.

```
class CalcFuncs (a) {
  deriv:: (a -> b) -> a -> b where Fractional(b);
  defint:: (a -> b) -> a -> a -> b where Fractional(b);
  indefint:: (a -> b) -> ((a -> b), a) where Fractional(b);

  defint (f,b,e) = g(e) - g(b)
                   where { (g,_) = indefint (f, zero); } ; }
where Fractional (a);

instance CalcFuncs (Phys (a)) {
  deriv = primitive "PhysicalDerivative";
  defint = primitive "PhysicalDefiniteIntegral";
  indefint = primitive "PhysicalIndefiniteIntegral"; } ;
```

4 In addition to the standard numeric functions, the impulse function is pre-defined as a `Funcsig`.

```
  impulse:: Funcsig;
  impulse = primitive "ImpulseFunction";
```

10.5.6 Random Functions

1 The functions defined in this section provide the capability to describe random processes and statistical information. The function `rseed` provides a random seed for each definition in the design hierarchy, while the function `random` provides a random number given another number.

```
  rseed:: definition -> Float;
  rseed = primitive "RandomSeed";

  random:: Float -> Float;
  random = primitive "RandomNumberGenerator";

  norm:: Physical -> Physical -> Physical -> Physical;
  norm (mu, sig, x) = exp ((- (x - mu)**2) / (2*sig**2));

  log_norm:: Physical -> Physical -> Physical -> Physical;
  log_norm (alpha, beta, x) = (alpha / (sqrt(2*pi))*x) *
                              exp(-0.5*(alpha * ln(x) + beta)**2);
end Standard;
```

10.6 MHDL Standard

1 Standard models include models and packages that provide general capabilities to the MHDL community. These include packages for describing the interface between MHDL models and VHDL models, and for specifying connector kinds, or "types", and attributes on the different kinds of connectors.

10.6.1 VHDL

1 The package `VHDL` describes the interface to a VHDL model. It contains definitions for the form of a VHDL signal and a VHDL driver.

2 An MHDL model uses a component that is described by a VHDL entity architecture pair by writing a model which attaches the name of the entity architecture pair, in the form of a string attribute `VHDL_entity`, to the structure that is associated with the component. The connector names must have the same names as, and the attributes should have compatible types with, the ports of the VHDL entity. The MHDL model should only define a value for the `[port]_value` attribute of a conection that represents a VHDL port. The VHDL model will provide an initial driver value for the attribute `[port]_driving` of any connector.

3 If an MHDL model is instantiated by a VHDL description, there is a VHDL entity that corresponds to the MHDL model (where the MHDL model serves as the architecture). The connectors of the MHDL model must have the same names as the ports of the VHDL entity, and the attributes should have a type compatible with the types of the VHDL ports of the same name. The MHDL model should only give a value to the `[port]_driving` attribute of its connectors. The MHDL model has the responsibility of providing an initial value to these drivers.

4 The attribute `[port]_value` is attached to any connector of mode `in_dir` or `inout_dir` on the VHDL port, where `[port]` is replaced with the port name. This represents the resolved value of the signal, as defined by the VHDL semantics ([10]). The attribute `[port]_driving` is attached to any connector of mode `out_dir` or mode `inout_dir` on the corresponding VHDL port, where `[port]` is replaced by the port name. This represents the driving value of the port, as defined by the VHDL semantics.

5 The relationship between the VHDL model and the MHDL model (from the MHDL side of things) is that of mutually recursive value definitions, where some of those value definitions exist in the VHDL model. Any implementation of the interface between the VHDL model and the MHDL model should be careful to avoid "posting" the new values to the ports until the time relationship between the two models warrents this posting. A VHDL model can be defined in this manner, as discussed in [2].

6 An MHDL model that is connected to a VHDL model has the obligation to define the attribute `[port]_name` for each `in` or `inout` child connector, where that attribute is of the appropriate type. It also has obligation to define the attribute `[port]_name` for each `inout` or `out` parent connector, where that attribute is of the appropriate type.

7 Except for VHDL physical types, the relationship between MHDL types and VHDL types is straightforward. For each VHDL physical type, the MHDL user should include a unit definition equal to the base unit of the VHDL physical type. The corresponding unit type, using `Integer` for the polymorphic name of the unit type, will match the VHDL physical type value. See the definition of `VHDL_signal`, below, for an example. Additional units can be added to correspond to the VHDL units, if desired.

```
package VHDL
  type VHDL_signal (a) = [ (Time, Int, a) ] ;
```

```
  attribute VHDL_entity: String;
end VHDL;
```

10.6.2 Connector Types

1 The following packages implement connector types in MHDL. Each attribute that is assumed to be of a connector type has certain attributes defined for it, and may also have some equations associated with it. If connectors exist in the same structure which are of mixed types, the structure must inherit all applicable packages and must type the connectors explicitly. In the case where all the connectors of the structure are of the same type, the appropriate structure package (using the name with `str` prefixed) can be inherited by the structure. This will type all connectors, and apply the equations and expectations as needed.

```
package electrical_attributes
    attribute excluded_names : [ String ] ;
    attribute connection_sets : [ [ String ] ] ;
end electrical_attributes;


package electrical
    includes electrical_attributes;
    for each connector C generate
        if C.ctype == electric   generate
            C.connection_sets = if child_conn_set(C) == [] then
                                     [sort(connection_set (C))]
                                 else
                                     [];
            C.excluded_names = if child_conn_set(C) == [] then
                                   []
                               else
                                   [full_name(C)];
        end generate ;
    end generate ;
    electrical.excluded_names  =
       join(map(snd,structure_conn_obj(electrical,excluded_names))) ++
       join(map(snd,structure_comp_obj(electrical,excluded_names)));
    electrical.connection_sets =
       join(map(snd,structure_conn_obj(electrical,connection_sets))) ++
       join(map(snd,structure_comp_obj(electrical,connection_sets)));
```

```
end electrical;

package electrical_str
    includes electrical_attributes;
    for each connector C generate
        C.ctype = electric;
        C.connection_sets = if child_conn_set(C) == [] then
                                [sort(connection_set (C))]
                            else
                                [];
        C.excluded_names = if child_conn_set(C) == [] then
                                []
                            else
                                [full_name(C)];
    end generate ;
    electrical_str.excluded_names  =
     join(map(snd,structure_conn_obj(electrical_str,excluded_names))) ++
     join(map(snd,structure_comp_obj(electrical_str,excluded_names)));
    electrical_str.connection_sets =
     join(map(snd,structure_conn_obj(electrical_str,connection_sets))) ++
     join(map(snd,structure_comp_obj(electrical_str,connection_sets)));
end electrical_str;

package electrical_root
    includes electrical_attributes;
    filter_list            :: [String] -> [[String]] -> [[String]];
    filter_list (fltr,lst) = map(filter(in_op(,fltr)),lst);

    nodup_list         :: [[String]] -> [[String]];
    nodup_list ([])    = [];
    nodup_list (l:-ls) = nodup_list1(l,ls);

    nodup_list1                      :: [String] -> [[String]] -> [[String]];
    nodup_list1 (x,[])               = [x];
    nodup_list1 ([],[]:-ls)          = nodup_list1([],ls);
    nodup_list1 ([],l:-ls)           = []:-nodup_list1(l,ls);
    nodup_list1 (x:-xs,(y:-ys):-ls) = if x == y then
                                          nodup_list1(x:-xs,ls)
                                       else
                                          (x:-xs) :- nodup_list1(y:-ys,ls);

    model_connection_sets :: [ [ String ] ];
```

```
    model_connection_sets = filter_list(excluded_names,
                                        nodup_list(sort(connection_sets)));

    kcl_check :: [[String]] -> time -> bool;
    kcl_check(c,t) = foldl(checkt,True,c)
      where {  checkt (f,l) = f && sumt(l) == 0.0 'amps' ;
               sumt(l) = foldl((+),0.0 'amps', map(appt,l));
               appt (sig) = Current_dimtoPhys(
                            (attribute_value(i,sig))(t)); }  ;
    KCL:  kcl_check(model_connection_sets) ;

    kvl_check :: [[String]] -> time -> bool;
    kvl_check (c,t) = foldl(checkt,True,c)
      where {  checkt (f,l) = f && alleqt(l);
               alleqt([]) = True;
               alleqt([x]) = True;
               alleqt(x:-xs) = foldl((&&),True,map(elemeq(x),xs));
               elemeq(y,z) = appt(y) == appt(z);
               appt (sig) = Potential_dimtoPhys(
                            (attribute_value(v,sig))(t)); } ;
    KVL:  kvl_check(model_connection_sets) ;
end electrical_root;

package signal_funcs
  count_outs:: [(String, Dir)] -> Int;
  count_outs ([]) = 0;
  count_outs ((n,dir):-xs) = (if dir == out_dir || dir == inout_dir
                              then 1 else 0) + count_outs (xs);
end signal_funcs;

package signal_flow
  includes signal_funcs;

  for each connector C generate
    if C.ctype==sigflow generate
      C_outs :: label;
      C_outs: count_outs (structure_conn_obj (C, dir)) +
               count_outs (child_conn_obj  (C, dir)) == 1;
    end generate;
  end generate;
end signal_flow;
```

```
package str_signal_flow
  includes signal_funcs;
  for each connector C generate
    C_outs :: label;
    C.ctype = sigflow;
    C_outs: count_outs (structure_conn_obj (C, dir)) +
             count_outs (child_conn_obj  (C, dir)) == 1;
  end generate;
end str_signal_flow;

package distributed
  for each connector C generate
    if C.ctype == phasor generate
     end generate;
  end generate;
end distributed;

package str_distributed
  for each connector C generate
    C.ctype = phasor;
  end generate;
end str_distributed;
```

10.6.3 Ground

The concept of a standard ground is defined in MHDL by two separate standard packages, `full_ground`, and `local_ground`. Each declares a ground connector, and connects the ground connectors of parents to children. The package `full_ground` specifies that the electrical potential of the ground pin is `0 'volts'`; the `local_ground` package does not. The inclusion of either one in the model containing a structure `electrical_breakdown` will add the connectors to that package.

```
package local_ground
  structure electrical_breakdown
    connectors ground;
    for each component C  generate
        connect ground, C.ground ;
    end generate;
  end electrical_breakdown;
end local_ground;
```

```
package full_ground
  includes local_ground;
  electrical_breakdown.ground.v = 0.0 'volts';
end full_ground;
```

Bibliography

[1] ISO Technical Committee 184 Subcommittee 4. Product data representation and exchange, part 42: Integrated resources: Geometric and topological representation. Technical Report ISO CD 10303-42, International Standards Organization, June 1991.

[2] David L. Barton. A functional characterization of elements of the vhdl simulation cycle. In *Using VHDL for Electronic Product Design*, pages 91–96. VHDL User's Group, April 1991.

[3] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proceedings of the ACM Conference on Lisp and Functional Programming*. ACM, June 1992.

[4] Luis Dumas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM, January 1982.

[5] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[6] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell. Technical Report YALEU/DCS/RR-777, Yale University Department of Computer Science, January 1992.

[7] Simon L. Peyton Jones and Phillip Wadler. Imperative functional programming. In *Proceedings of the 20th Annual Symposium on Principles of Programming Languages*. ACM, 1993.

[8] United States Army LABCOM-ETDL. MHDL requirements document. Technical Report Edition 1, United States Army LABCOM-ETDL, August 1991.

[9] Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating Point Arithmetic*. Number IEEE Std 754. Institute of Electrical and Electronic Engineers, New York, New York, 1985.

[10] Institute of Electrical and Electronic Engineers. *IEEE Standard VHDL Language Reference Manual.* Number IEEE Std 1076. Institute of Electrical and Electronic Engineers, New York, New York, 1987.

[11] International Standards Organization. *SI units and recommendations for the use of their multiples and of certain other units.* Number ISO 1000. International Standards Organization, Geneva, Switzerland, 1981.

[12] International Standards Organization. *-Bit Single Byte Coded Graphic Character Sets — Part 1: Latin Alphabet No. 1.* Number ISO 8859-1. International Standards Organization, Geneva, Switzerland, 1987.

[13] Phillip Wadler. Comprehending monads. In *Proceedings of the Conference on Lisp and Functional Programming.* ACM, 1990.

[14] Phillip Wadler. The essence of functional programming. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages.* ACM, 1992.

[15] Phillip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.

Appendix A

Syntax Summary

Grammar Rules:

```
absolute_configuration ::=
    [ use MODEL_OR_STRUCTURE_OR_CONFIGURATION_name ]
    { configuration_clause ; }

abstract_property_definition ::=
    property_kind abstract_property_name_list : type_specification

abstract_property_name_list ::=
    ABSTRACT_PROPERTY_simple_name
        { , ABSTRACT_PROPERTY_simple_name }
alternative ::=
    general_alternative
  | guarded_alternative

alternatives ::=
    alternative ; { alternative ; }

any_e ::=
    E
  | e

any_sign ::=
    +
```

```
  | -

argument ::=
    [ expression ]

arithmetic_sequence ::=
    '[' expression [ , expression ] .. [ expression ] ']'

as_pattern ::=
    variable @ simple_pattern

ascii_escape ::=
    ^ control_character
  | NUL | SOH | STX | ETX | EOT | ENQ | ACK
  | BEL | BS  | HT  | LF  | VT  | FF  | CR
  | SO  | SI  | DLE | DC1 | DC2 | DC3 | DC4
  | NAK | SYN | ETB | CAN | EM  | SUB | ESC
  | FS  | GS  | RS  | US  | SP  | DEL

association_definition ::=
    associate name , name { , name }

base ::=
    digits

based_floating_point_part ::=
    base # extended_digits . extended_digits #

based_integer_part ::=
    base # extended_digits #

basic_identifier ::=
    letter { [ underline ] letter_or_digit }

case_configuration ::=
    case expression of '{'
        component_alternative ; { component_alternative ; }
    else
        absolute_configuration ; '}'

case_expression ::=
    case expression of '{' alternatives '}'
```

```
character_literal ::=
    '' single_character ''

class_body ::=
    '{' class_definition_list  '}'

class_constraint ::=
    instance_spec

class_definition ::=
    class_header [ class_body ] [ where context ]

class_definition_list ::=
    function_definition ; { function_definition ; }

class_header ::=
    class CLASS_simple_name class_type_list

class_list ::=
    CLASS_name { , CLASS_name }

class_type_list ::=
    ( type_specification { , type_specification } )

column_header ::=
    columns STRING_expression { '|' STRING_expression } ;

column_name_list ::=
    names identifier { '|' identifier } ;

complex_configuration ::=
    for COMPONENT_OR_STRUCTURE_name
        configuration_alternative
    end [ COMPONENT_OR_STRUCTURE_name ]

component_alternative ::=
    pattern -> absolute_configuration

component_definition ::=
    COMPONENT_simple_name :: [ STRUCTURE_OR_MODEL_name ]
        [ [ inclusion_list ]
```

```
        [ definitions definition_list ]
    end [ COMPONENT_simple_name ] ]

component_list ::=
    components
        component_definition ; { component_definition ; }
    end components

conditional_expression ::=
    if expression then
        expression
    else
        expression

configuration_alternative ::=
    absolute_configuration
  | if_configuration
  | case_configuration

configuration_body ::=
    configuration_clause ; { configuration_clause ; }

configuration_clause ::=
    simple_configuration
  | complex_configuration

configuration_definition ::=
    configuration CONFIGURATION_simple_name
        configuration_body
    end [ CONFIGURATION_simple_name ]

connection_list ::=
    [ label ] connect
        connector_specification { , connector_specification }

connector_clause ::=
    connectors [ connector_name_list ]

connector_name_list ::=
    CONNECTOR_simple_name { , CONNECTOR_simple_name }

connector_specification ::=
```

```
        CONNECTOR_OR_CONNECTION_LIST_name

    constraint_definition ::=
        constraint CONSTRAINT_simple_name : BOOLEAN_expression
            [ report STRING_expression ]

    constructed_type ::=
        constructor_list [ with `{` constructor_definitions `}` ]

    constructor ::=
        name

    constructor_definitions ::=
        value_definition ; { value_definition ; }

    constructor_list ::=
        type_constructor { '|' type_constructor }

    context ::=
        class_constraint
      | '{' class_constraint { , class_constraint } '}'

    control_character ::=
        upper_case_letter
      | @
      | [
      | \
      | ]
      | ^
      | &

    conversion_spec ::=
        RATIONAL_expression
      | FUNCTION_expression inverse FUNCTION_expression

    data_table ::=
        '{{' { row } '}}'

    decimal_floating_point_part ::=
        digits . digits

    decimal_integer_part ::=
```

```
    digits

definition_designator ::=
    model
  | structure
  | constraint
  | equation
  | function
  | connector
  | component
  | property ( type_specification )
  | variable ( type_specification )
  | label
  | definition
  | type

definition_list ::=
    structure_item ; { structure_item ; }

derivation_clause ::=
    deriving ( class_list )

derivative ::=
    derivative ( identifier ) ( FUNCTION_expression )

derivative_or_integral ::=
    derivative
  | integral

designation ::=
    ' designation_indicators '

designation_indicators ::=
    NUMERATOR_indicators [ '/' DENOMINATOR_indicators ]
  | ( designation_indicators )

digits ::=
    digit { digit }

dimension_definition ::=
    dimension DIMENSION_identifier
        [ = DIMENSION_designation ]
```

```
        [ DIMENSION_variations_clause ]

else_generate ::=
    else generate
      generate_body

else_if_generate ::=
    else if BOOLEAN_expression
      generate generate_body

equation_definition ::=
    [ label ] expression

escaped_character ::=
    \ special_character
  | \ ascii_escape
  | \ digits
  | \ o digits
  | \ x extended_digits

escaped_identifier ::=
    ` graphic_character { graphic_character } `

explicit_irrefutable_pattern ::=
    ~ simple_pattern

exponent ::=
    any_e [ any_sign ] digits

exposure_clause ::=
    [ exposure_list from ] MODEL_OR_PACKAGE_name

exposure_item ::=
    DEFINITION_name [ as DEFINITION_simple_name ]

exposure_list ::=
    exposure_item { , exposure_item }

expression ::=
    function_application
  | operator_application
  | lambda_abstraction
```

```
  | let_expression
  | conditional_expression
  | case_expression
  | typed_expression
  | structure_function_creation
  | derivative_or_integral
  | simple_expression

extended_digit ::=
    digit
  | letter

extended_digits ::=
    extended_digit { extended_digit }

field_list ::=
    [ FIELD_simple_name :: ] type_component
        { , [ FIELD_simple_name :: ] type_component }

floating_point_literal ::=
    decimal_floating_point_part [ exponent ]
  | based_floating_point_part [ exponent ]

for_generate_definition ::=
    for_generate_header
      generate_body
    end generate

for_generate_header ::=
    for each generate_object_kind
      GENERATE_simple_name
      [ in generate_list ]
      generate

function_application ::=
    simple_expression ( argument  { , argument } )

function_definition ::=
    name_definition
  | value_definition

function_designation ::=
```

```
      FUNCTION_name ( pattern { , pattern } )
    | pattern operator pattern

  gap ::=
      \ white_space \

  general_alternative ::=
      pattern -> expression [ where nested_definitions ]

  generate_body ::=
      generate_item ; { generate_item ; }

  generate_definition ::=
      if_generate_definition
    | for_generate_definition
    | table_generate_definition

  generate_item ::=
      structure_item
    | structure_definition

  generate_list ::=
      STRUCTURE_OR_MODEL_OR_COMPONENT_OR_PACKAGE_name
    | LIST_expression

  generate_object_kind ::=
      component
    | connector
    | structure
    | member
    | connection set

  generator ::=
      '[' expression '|' qualifier { , qualifier } ']'

  guard ::=
      '|' expression

  guard_pattern ::=
      guard -> expression [ guard_pattern ]

  guarded_alternative ::=
```

```
    pattern guard_pattern [ where nested_definitions ]

guarded_expression_list ::=
    guard = expression [ guarded_expression_list ]

hiding_clause ::=
    MODEL_OR_PACKAGE_name [ except hiding_list ]

hiding_list ::=
    DEFINITION_name { , DEFINITION_name }

identifier ::=
    basic_identifier
  | escaped_identifier

if_configuration ::=
    if BOOLEAN_expression then
       absolute_configuration
    { else if BOOLEAN_expression then
       absolute_configuration }
    else absolute_configuration

if_generate_definition ::=
    if_generate_header
      { else_if_generate }
      [ else_generate ]
    end generate

if_generate_header ::=
    if BOOLEAN_expression
      generate  generate_body

inclusion_list ::=
    includes visibility_item ;
        { includes visibility_item ; }

indicator ::=
    DIMENSION_OR_UNIT_name [ ^ indicator_exponent ]
  | ( indicator )

indicator_exponent ::=
    [-] integer_literal [ '//' integer_literal ]
```

```
    | ( indicator_exponent )

indicators ::=
    indicator { '*' indicator }
  | ( indicators )

instance_body ::=
    value_definition ; { value_definition ; }

instance_definition ::=
    instance_header
        [ '{' instance_body '}' ]
        [ where context ]

instance_header ::=
    instance instance_spec

instance_spec ::=
    CLASS_name ( type_specification { , type_specification } )

integer_literal ::=
    decimal_integer_part [ exponent ]
  | based_integer_part [ exponent ]

integer_or_float ::=
    integer_literal
   |   floating_point_literal

integral ::=
    integral ( identifier )
        [ ( REAL_expression , REAL_expression ) ]
        ( FUNCTION_expression )
label ::=
    identifier ':'

lambda_abstraction ::=
    \ ( pattern { , pattern } ) -> expression

let_expression ::=
    let nested_definitions in expression

letter ::=
```

```
    upper_case_letter
  | lower_case_letter

letter_or_digit ::=
    letter
  | digit

library_list ::=
    libraries LIBRARY_simple_name
        { , LIBRARY_simple_name } ;

list ::=
    '[' [ expression { , expression } ] ']'

list_pattern ::=
    '[' [ pattern {, pattern } ] ']'

literal ::=
    integer_literal
  | floating_point_literal
  | character_literal
  | string_literal

model ::=
    model MODEL_simple_name
        [ library_list ]
        [ inclusion_list ]
        [ model_body ]
    end [ MODEL_simple_name ] ;

model_body ::=
    model_item ; { model_item ; }

model_item ::=
    structure_definition
  | structure_item
  | class_definition
  | instance_definition
  | type_definition
  | dimension_definition
  | unit_definition
  | multiplier_specification
```

```
multiplier_specification ::=
  multiplier MULTIPLIER_identifier
    = INTEGER_OR_FLOAT_literal [ MULTIPLIER_variations_clause ]

name ::=
    simple_name
  | selected_name

name_definition ::=
    [ coercion ] named_values :: type_specification
         [ where context ]

named_values ::=
    NAMED_VALUE_simple_name {, NAMED_VALUE_simple_name }

nested_definition ::=
    name_definition
  | value_definition

nested_definitions ::=
    '{' nested_definition ; { nested_definition ; } '}'

null_expression ::=
    ()

null_pattern ::=
    ()

operator ::=
    operator_symbol
  | `` FUNCTION_name ``

operator_application ::=
    expression operator expression
  | - expression

package ::=
    package PACKAGE_simple_name
        [ library_list ]
        [ inclusion_list ]
        [ model_body ]
```

```
    end [ PACKAGE_simple_name ] ;

parenthesized_expression ::=
    ( expression )

pattern ::=
    pattern [ CONSTRUCTOR_operator pattern ]
  | - integer_or_float
  | simple_pattern
  | constructor ( pattern { , pattern } )

physical_value ::=
    REAL_AND_SCALE_expression UNIT_designation

polymorphic_name_list ::=
    ( POLYMORPHIC_simple_name { , POLYMORPHIC_simple_name } )

primitive_type_definition ::=
    type type_header

primitive_value ::=
    primitive STRING_expression

property_kind ::=
    attribute
  | signal

property_name_list ::=
    ( PROPERTY_name { , PROPERTY_name } )

qualifier  ::=
    pattern <- expression
  | expression

row ::=
    table_entry { '|' table_entry } ;

selected_name ::=
    { identifier . } identifier . simple_name

simple_configuration ::=
    for COMPONENT_OR_STRUCTURE_name
```

```
        use MODEL_STRUCTURE_OR_CONFIGURATION_name

simple_expression ::=
    NAMED_VALUE_name
  | literal
  | null_expression
  | parenthesized_expression
  | tuple
  | list
  | arithmetic_sequence
  | generator
  | physical_value
  | primitive_value

simple_name ::=
    identifier
  | ( operator )

simple_pattern ::=
    NAMED_VALUE_variable
  | as_pattern
  | constructor
  | literal
  | wildcard_pattern
  | null_pattern
  | ( pattern )
  | tuple_pattern
  | list_pattern
  | explicit_irrefutable_pattern

single_character ::=
    graphic_character
  | escaped_character
  | gap

special_character ::=
    a | b | f | n | r | t | v | \ | " | ' | &

specified_type_definition ::=
    type type_header = type_specifier [ derivation_clause ]
        [ where context ]
```

```
string_literal ::=
    " { single_character } "

structure_definition ::=
    structure STRUCTURE_simple_name
        [ inclusion_list ]
        [ structure_element_list ]
    end [ STRUCTURE_simple_name ]

structure_element ::=
    structure_item
  | connector_clause
  | component_list

structure_element_list ::=
    structure_element ; { structure_element ; }

structure_function_creation ::=
    property_name_list <== STRUCTURE_name property_name_list

structure_item ::=
    connection_list
  | abstract_property_definition
  | name_definition
  | value_definition
  | equation_definition
  | constraint_definition
  | association_definition
  | configuration_definition
  | generate_definition
  | table_definition

table_definition ::=
    table TABLE_simple_name
        [ table_header ]
        [ column_header ]
        column_name_list
        data_table
    end [ TABLE_simple_name ]

table_entry ::=
    white_space true_table_entry white_space
```

```
table_generate_definition ::=
    table_generate_header
      generate_body
    end generate

table_generate_header ::=
    for each row
    [ GENERATE_simple_name ] in
    table TABLE_name generate

table_header ::=
    title STRING_expression ;

true_table_entry ::=
    { graphic_character }

tuple ::=
    ( expression , expression { , expression } )

tuple_pattern ::=
    ( pattern , pattern {, pattern } )

type_component ::=
    CONSTRUCTOR_simple_name ( type_component { , type_component } )
  | definition_designator
  | TYPE_name
  | POLYMORPHIC_simple_name
  | ( )
  | ( type_specification )
  | ( type_list )
  | '[' type_specification ']'
  | '[' ']'
  | '[' ']' ( type_specification )
  | ( -> )
  | ( -> ) ( type_list )
  | ( , { , } )
  | ( , { , } ) ( type_list )

type_constructor ::=
    CONSTRUCTOR_simple_name [ ( field_list ) ]
  | type_component CONSTRUCTOR_operator type_component
```

```
type_definition ::=
    primitive_type_definition
  | specified_type_definition

type_header ::=
    TYPE_simple_name [ polymorphic_name_list ]

type_list ::=
    type_specification , type_specification
        { , type_specification }

type_specification ::=
    type_component [ -> type_specification ]

type_specifier ::=
    type_specification
  | constructed_type

typed_expression ::=
    ( expression :: type_specification [ where context ] )

underline ::=
    _

unit_conversion ::=
    conversion conversion_spec UNIT_name

unit_definition ::=
    unit UNIT_identifier of DIMENSION_name
        [ = UNIT_designation ]
        [ unit_conversion ]
        [ UNIT_variations_clause ]

value_definition ::=
    [ label ] value_definition_target = expression
        [ where_clause ]
  | [ label ] value_definition_target guarded_expression_list
        [ where_clause ]

value_definition_target ::=
    function_designation
```

```
    | [ default ] pattern

variable ::=
    name

variations_clause ::=
    variations identifier { , identifier }

visibility_item ::=
    exposure_clause
  | hiding_clause

where_clause ::=
    where nested_definitions

wildcard_pattern ::=
    _
```

Appendix B

Glossary

abstract property A name that may be attached to other, unique names in a design hierarchy. An abstract property is used, in conjunction with a hierarchical name, to identify a value attached to parts of the MHDL description. See 5.1 [Abstract Properties], page 55, for more information.

applied A function that is provided arguments and used to obtain a return value is said to be applied. See 6.2.1.1 [Functions], page 67, for more information.

apply to A function binding applies to a function if it defines the value of that function for some combination of the functions inputs, where this combination is specified by a pattern. See 6.2.1 [Function Bindings], page 65, for more information.

apply A function that is provided arguments and used to obtain a return value is said to be applied; such a use is a function application. See 7.1 [Function Applications], page 81, for more information.

argument type The argument type of a function is the type of the value that must be provided to the function in order for it to be applied. See 6.2.1.1 [Functions], page 67, for more information.

arguments Values that are provided to a function in order for that function to be evaluated, or applied, are called arguments. Names that stand for those values in the function definition are sometimes called formal arguments. See 6.2.1.1 [Functions], page 67, for more information.

arity The arity of a function is the maximum number of arguments that the function takes in order to be applied. See 6.2.1 [Function Bindings], page 65, for more information.

array An array is a sequential grouping of values that may be selected by providing a value, called the index, to the array application. Index values must be sequential. See 10.3.3 [Array], page 155, for more information.

as pattern In complex patterns, the user sometimes wishes to give a name to a portion of the pattern that is matched. This is indicated by using the at sign `@`, and such a pattern is referred to as an "as pattern". See 6.3 [Patterns], page 72, for more information.

assignment An assignment is a value definition that gives a value to a property. See 5.2 [Property Value Definitions], page 57, for more information.

associated property Any property that is associated to another property is called an associated property; thus, two properties that are associated are associated properties of each other. This has implications for the elaboration process. See 3.4 [Elaboration], page 26, for more information.

associated Two names are associated if the values attached to one name can be read by another name as if they were attached to that name. See 5.3 [Associations], page 58, for more information.

base set The base set of an expression or function application is the set of expressions that appear as arguments for that expression or function application. See 4.3 [Type Coercions], page 51, for more information.

base unit A unit for a dimension that is not given a conversion, and which therefore in some sense defines the dimension, is called the base unit for the dimension. The dimension itself can be derived. See 4.2.12.2 [Unit Definition], page 47, for more information.

based literal A literal that is expressed within a numeric system based on a number other than 10 (2, 8, or 16) is a based literal. See 9.5 [Literals], page 110, for more information.

basic dimension A dimension which is not defined in terms of other dimensions, but which stands on its own as primary or primitive, is a basic dimension. See 4.2.12.1 [Dimension Definition], page 44, for more information.

basic unit A basic unit is the base unit of a basic dimension. See 4.2.12.2 [Unit Definition], page 47, for more information.

binding A name is given a value as a result of being matched to some expression, or part of an expression, where the name is contained in a pattern. The result of matching the pattern to the value resulting from the evaluation of the expression gives a value to the name; this process is called binding. See 6 [Named Values], page 61, for more information.

bodies The expressions that appear in the alternatives of a case expression, following the `->` symbols, are the bodies of the case expression. See 7.6 [Case Expressions], page 85, for more information.

children The children of a part are those parts which are associated with the components of the structures in the part, and which are indicated by those components. See 3.3 [Design Hierarchies], page 25, for more information.

coercion functions A coercion function is a function that is explicitly designated as the means by which values of one type are changed to values of another type. Under some circumstances, coercion function applications are automatically inserted into an expression. See 4.3 [Type Coercions], page 51, for more information.

component definitions A component definition defines a subsystem, or a subsection, of an analog or microwave design. Hierarchical decomposition is defined by components, and connecting those components together. See 3.1 [Structure Definitions], page 17, for more information.

component A component is a subsystem, or a subsection, of an analog or microwave design or device. Hierarchical decomposition is defined by components, and by connecting those components together. See 3.1.1 [Components], page 19, for more information.

composite type A composite type is a set of values that are made up of collections of other values. Sets and arrays are examples. See 10.3 [Composite Types], page 148, for more information.

configuration A configuration gives information about the relationship between components and models; specifically, it gives the structure indicated for each component, thereby specifying the model to be used to create the part for the component. See 3 [Structure], page 17, for more information.

configured structure A configured structure is a structure that appears in a configuration, and thus that has the structures for its components given by the configuration. See 3.2 [Configurations], page 22, for more information.

connector A connector is a name inside a structure to which properties may be attached. Connectors are associated with other connectors by connection lists, thereby specifying the flow of information through the design. See 3.1.2 [Connectors], page 21, for more information.

constraint A constraint specifies a condition that must be true in the design. If a constraint is not true, or if it is violated, an error message may be produced. See 6.4 [Constraints], page 75, for more information.

constructed pattern A constructed pattern is a pattern that is built up using one or more constructors, including constructor functions, constructor operators, lists, and tuples. See 6.3 [Patterns], page 72, for more information.

constructor A constructor is a name that identifies a value of a specific type, called a constructed type. A constructor also serves as a function that can build values of that type, and a recognizer that can be used in a pattern to identify values of that type. See 4.1.1 [Specified Types], page 30, for more information.

containing structure A structure is the containing structure of the connectors that are defined within it. See 3.1.2 [Connectors], page 21, for more information.

correct A design is correct if all of the constraints and equations in the design are met; or, if for all the possible values that can be given to the expressions in the equations and constraints, the expressions evaluate to True. See 6.5 [Equations], page 76, for more information.

created property A property that is created during the process of elaborating a specific part is called a created property of that part. See 3.4 [Elaboration], page 26, for more information.

creating an entity During elaboration, names to which properties are attached are identified, and any processing necessary for those names to be used as labels for unique properties is performed. This may or may not include reserving computer memory and creating symbol table entries. This process is called creating an entity. See 3.4 [Elaboration], page 26, for more information.

creating the property During elaboration, unique names and space for values are created for each property in a part. This process is called creating the property. See 3.4 [Elaboration], page 26, for more information.

currying A function that takes a number of arguments may be called with a smaller number of arguments, which then returns a function which in turn takes the remaining arguments (not provided in the first call) before giving an answer. Invoking the original function with a subset of the full set of arguments is called currying. See 6.2.1.1 [Functions], page 67, for more information.

default pattern binding A default pattern binding, or default value, or simply a default, is a value that may be overridden by another pattern binding in a value definition. Such pattern bindings are explicitly set out with the keyword `default`. See 6.2.2 [Pattern Bindings], page 69, for more information.

definite integral A definite integral is an integral which gives an explicit region over which the integration is to be performed; in MHDL, this is a lower and upper bound that describe a segment of the real number line. See 7.9 [Derivatives and Integrals], page 89, for more information.

definition set During the elaboration of a generate definition, the definitions in a generate statement are in some cases duplicated for each of a number of definitions that occur locally; for example, for each connector in a structure. The definitions for which the generated definitions are duplicated is called the definition set of the generate statement. See 2.3 [Generate Definitions], page 9, for more information.

derivative A derivative is a mathematical expression that defines the rate of change of a function at a given value for one of its arguments. See 7.9 [Derivatives and Integrals], page 89, for more information.

derived dimension A derived dimension is a dimension which is defined by some combination of other dimensions; speed, for example, which is defined in terms of length and time. See 4.2.12.1 [Dimension Definition], page 44, for more information.

derived from A model which names another model in its inclusion list is said to be derived from the model it names. See 2 [Models], page 5, for more information.

derived instance definitions For some specific classes named in package `Standard`, instances need not be given explicitly; instead, they are named in the derivation clause of the type definition, and the instances are generated automatically. These instances are derived instance definitions. See 4.1.1 [Specified Types], page 30, for more information.

derived unit A derived unit is a unit which is given in terms of another unit; for example, an inch is .0254 meters. See 4.2.12.2 [Unit Definition], page 47, for more information.

description unit A description unit is a model or a package. Description units are grouped together in libraries; see 2.6 [Libraries], page 15, for more information.

design hierarchy A design hierarchy is a tree of parts, each one of which is created from a model. Each branch of the tree represents a component, and the part created by elaborating the component. See 3 [Structure], page 17, and see 3.3 [Design Hierarchies], page 25, for more information.

designation A designation is an identification of a unit, a dimension, or an MHDL structure that is being passed to a primitive function. See 4.2.12.1 [Dimension Definition], page 44, for more information.

dimension designation A dimension designation is an algebraic specification of a dimension, using other dimensions to specify the contents of the described dimension. See 4.2.12.1 [Dimension Definition], page 44, for more information.

dimension type Each dimension has an associated type, which described numeric quantities of dimension. See 4.2.12.1 [Dimension Definition], page 44, for more information.

dimension A dimension is a physical quantity, such as time or length. See 4.2.12.1 [Dimension Definition], page 44, for more information.

diverge In the process of pattern matching, conditions may arise in which the continued attempt at matching the pattern is not defined. In these cases, the pattern matching process is said to diverge. See 6.3 [Patterns], page 72, for more information.

element type An array is a collection of values of a single type; that type is the element type of the array. See 10.3.3 [Array], page 155, for more information.

entity An entity is the internal representation of a name to which a property is attached. See 3.4 [Elaboration], page 26, for more information.

enumeration type An enumeration type is a type consisting of a series of constructors that have no arguments. These constructors represent the values of the enumeration type. See 4.1.2.2 [Enumeration Types], page 37, for more information.

equation An equation is a boolean expression that places constraints on the values that the properties of a description may assume. See 6.5 [Equations], page 76, for more information.

exceptions An exception is a specific value that represents an error condition. When used with a specific style of function definition, exceptions may propagate without explicit intervention from the user. See 10.3.4 [Exceptions], page 161, for more information.

exposed names See 2.1 [Inclusion Lists], page 7, for more information.

fail A pattern is used to describe the physical structure of a value, where the parts of the pattern are matched against the parts of the value. If such a match cannot be derived, the pattern is said to diverge. See 6.3 [Patterns], page 72, for more information.

format effectors A format effector is a character that occurs in the whitespace of a piece of MHDL text. See 9.1 [Character Set], page 107, for more information.

function binding A function binding is a value definition whose target is a function designation; i.e., a value definition that partially or wholly sets the value of a function. See 6.2 [Value Definitions], page 63, for more information.

function A function is a named value that can be used to transform values into other values; when the arguments of a function are provided, and the function is invoked with those arguments, a value is given as a result. See 6.2.1 [Function Bindings], page 65, and see 6 [Named Values], page 61, for more information.

generate definition A generate definition is an MHDL mechanism for repeating a specified group of definitions in one of a number of patterns. See 2.3 [Generate Definitions], page 9, for more information.

generated definitions The definitions duplicated by a generate definition are called the generated definitions. See 2.3 [Generate Definitions], page 9, for more information.

generation scheme A qualifier in a generator that produces a number of values, which may or may not be filtered, is called a generation scheme. See 7.10.7 [Generators], page 94, for more information.

generator A generator is an expression that produces one or a number of values that are members of a monad, using a specific set of rules. See 7.10.7 [Generators], page 94, for more information.

graphic characters A graphic character is a character that has meaning in MHDL beyond whitespace; intuitively, it is a character that prints. See 9.1 [Character Set], page 107, for more information.

guard A guard is an expression that filters the elements produced by an expression in a generator. See 7.10.7 [Generators], page 94, for more information.

guarded value definition A guarded value definition is a value definition given as a series of conditions followed by the expression to be used if the expression evaluates to true. See 6.2 [Value Definitions], page 63, for more information.

hierarchical name A hierarchical name identifies a property, structure, connector, constraint, label, part, or component in a design hierarchy. See 8.4 [Hierarchical Names], page 102, for more information.

included unit An included unit is a unit that is named in an inclusion list. See 8.6 [Visibility], page 104, for more information.

including unit An including unit is a unit that names another unit in its inclusion list. See 8.6 [Visibility], page 104, for more information.

inclusion list An inclusion list is a series of description unit names whose definitions are included in the description unit that contains the inclusion list. See 2 [Models], page 5, for more information.

indefinite integral An indefinite integral is an integral which specifies no space over which the integration is to be performed. See 7.9 [Derivatives and Integrals], page 89, for more information.

indicated structure A component is matched to a specific structure; this is the indicated structure for the component. See 3.1.1 [Components], page 19, for more information.

indicators An indicator is a dimension or unit name that is used to specify another dimension or unit. See 4.2.12.1 [Dimension Definition], page 44, for more information.

infix operator An infix operator names a function by occupying the space between the first argument of the function and the other arguments of the function; usually there are only two arguments. See 6.2.1.1 [Functions], page 67, for more information.

inherit A model or package that names a package in its inclusion list is said to inherit the package. See 2 [Models], page 5, for more information.

inheritance The process of including the definitions of a package into a model or package that names the included package in its inclusion list is called inheritance. See 2 [Models], page 5, for more information.

input list The definition of a structure function includes a list of the properties that will serve as the inputs of the function so created; this is the input list of the structure function. See 7.8 [Structure Functions], page 88, for more information.

instances An instance is a series of value definitions that give the meaning of values which are defined in a class for a specific type. See 4.1 [Type Definitions], page 30, for more information.

integral An integral is an expression that serves to represent a mathematical integral. See 7.9 [Derivatives and Integrals], page 89, for more information.

irrefutable pattern An irrefutable pattern is a pattern whose matching is delayed until a value bound in the pattern is actually required for another expression. Matching an irrefutable pattern thus always succeeds. See 6.3 [Patterns], page 72, for more information.

lazily evaluated A pattern match may be delayed until a value which is bound by the pattern is required for another calculation. In this case, the pattern is said to be lazily evaluated. See 6.3 [Patterns], page 72, and see 6.2.2 [Pattern Bindings], page 69, for more information.

less coerced An expression is less coerced than another expression if it includes fewer automatically inserted invocations of coercion functions. See 4.3 [Type Coercions], page 51, for more information.

libraries A library is a named collection of description units. See 2.6 [Libraries], page 15, for more information.

linear A collection of patterns is linear if no variable appears more than once in the collection of patterns. See 6.2.1 [Function Bindings], page 65, and see 6.3 [Patterns], page 72, for more information.

LRM This is an acronym for the MHDL Language Reference Manual. See 1 [Introduction], page 1, for more information.

matched against Names are given values in MHDL when a pattern is matched against a value. See 6.3 [Patterns], page 72, for more information.

matched lazily An irrefutable pattern is not matched against a value until the value associated with a name in the pattern is required; thus, the irrefutable pattern is matched lazily. See 6.3 [Patterns], page 72, for more information.

method A method is a function whose name definition occurs in a class definition. See 4.1.3 [Type Classes], page 37, for more information.

MHDL This is an acronym for the MIMIC Hardware Description Language, a language for the description of analog and microwave hardware. See 1 [Introduction], page 1, for more information.

minimally coerced An expression is minimally coerced if it includes the same number or fewer automatically inserted invocations of type coercion functions than any other expression which is type correct in the same context. See 4.3 [Type Coercions], page 51, for more information.

model A model is a collection of definitions that represents a component or subsystem in an analog or microwave design. See 2 [Models], page 5, for more information.

monad A monad is a type which is a member of the class `Monad`, and for which the functions result and bind are defined so that they obey certain laws. See 10.3.1 [Monad], page 148, for more information.

multiplier A multiplier is a prefix to a unit name that specifies a multiplicative factor of that unit. See 4.2.12.3 [Multiplier Specification], page 49, for more information.

name definition A name definition identifies a name that will stand for a value, and gives a type specification for that value. See 6 [Named Values], page 61, for more information.

named value A named value is a value that can be represented by a name in an expression. See 6 [Named Values], page 61, for more information.

names A name is an identifier or a complex series of identifiers that uniquely identifies a piece of the MHDL description — a property, for example. See 8 [Names], page 99, for more information.

nested comments Comments in MHDL may be nested inside other comments, so that when the comment brackets are removed from the outer comment, the inner comment remains a comment. See 9.8 [Comments], page 116, for more information.

operation An operation is a function that is defined inside a class. See 4.1.3 [Type Classes], page 37, for more information.

ordinary comments An ordinary comment is a comment that extends for all or part of a single textual line, and is marked off by two hyphens (--). See 9.8 [Comments], page 116, for more information.

output list The definition of a structure function includes a list of the properties that will serve as the outputs of the function so created; this is the output list of the structure function. See 7.8 [Structure Functions], page 88, for more information.

overloaded A value whose name definition occurs in a class is said to be overloaded, in the sense that a use of that name may indicate one of a number of different values defined in different instances of the class. See 4.1.3 [Type Classes], page 37, for more information.

package A package is a collection of related definitions that is included in models (and other packages) for the purpose of encapsulation. See 2.2 [Packages], page 9, for more information.

parent A part which contains several components, each of which is associated with the part created by elaborating those components respectively, is the parent of those parts. See 3.3 [Design Hierarchies], page 25, for more information.

part A part is an instance of a model, created either as the root of a design hierarchy or as a result of the elaboration of a component in another part. See 3.3 [Design Hierarchies], page 25, for more information.

partially applied A function is partially applied if a subset of the required arguments is provided to the function application; the function is then said to be curried. See 6.2.1.1 [Functions], page 67, for more information.

pattern binding A pattern binding is a value definition that gives a value to a name in the pattern. See 6.2 [Value Definitions], page 63, for more information.

pattern matching Pattern matching is the process of applying a value to a pattern in order to give values to the names in the pattern. See 6 [Named Values], page 61, for more information.

pattern A pattern is a "structural expression" that describes the structure of a value, and against which a value can be matched. See 6.3 [Patterns], page 72, for more information.

polymorphic type A polymorphic type is a type that contains polymorphic names which act as placeholders for types that may be provided later, in a specialization of the polymorphic type. See 4.1.1 [Specified Types], page 30, for more information.

primary name The primary name of a unit or dimension is the name that occurs within the definition of that unit or dimension by which the unit or dimension will be normally referred; variations on the primary name may be given for convenience. See 4.2.12.2 [Unit Definition], page 47, for more information.

principal type The principal type of an expression is the type to which the value represented by the expression belongs; the principal type of each expression must be appropriate for the context in which it appears. See 7 [Expressions], page 79, for more information.

properties A property is a value which is attached to a name in the description, and which therefore represents information associated with a particular piece or portion of the design represented by the design hierarchy. See 5 [Properties], page 55, for more information.

property value definition A property value definition is a value definition whose pattern consists of a single property name; this is also known as an assignment. See 5.2 [Property Value Definitions], page 57, for more information.

qualified name A qualified name is a name that identifies a construct in the description in some other description unit, in some other model or package. See 8.3 [Qualified Names], page 101, for more information.

record name A record name consists of a name that identifies a value of record type, followed by a field name that identifies a part of the record value. See 8.2 [Record Names], page 100, for more information.

refutable A refutable pattern is a pattern that is matched as soon as it is encountered in the process of pattern matching, rather than waiting for a name in the pattern to be required in a computation. See 6.3 [Patterns], page 72, for more information.

result type The type of the value that results from a function application is the result type of the function. See 6.2.1.1 [Functions], page 67, for more information.

root The root of a design hierarchy is the root of the tree of parts of which the design hierarchy consists. The root is the only part of the design hierarchy that is not associated with a component during the process of elaboration. See 3.3 [Design Hierarchies], page 25, for more information.

satisfied An equation is satisfied if it evaluates to true for all possible inputs, given the values that exist in the named values of the description. See 6.5 [Equations], page 76, for more information.

signal types A signal type is a function type whose two arguments are both physical types. See 4 [Types and Classes], page 29, for more information.

simple pattern binding A simple pattern binding consists of a pattern and an expression, without guards. See 6.2.2 [Pattern Bindings], page 69, for more information.

specialization A type is a specialization of a polymorphic type if it is obtained by the substitution of types for some or all of the polymorphic names of the more general type. See 4.1.1 [Specified Types], page 30, for more information.

structural function A structural function is a function that operates on various parts of the design hierarchy; a structural function is always defined by a primitive value. See 10.5.4 [Structural Functions], page 172, for more information.

structure definition A structure definition defines the division of some part of an analog or microwave design into subcomponents, which are then wired together by connecting up the connectors of the various components. See 3.1 [Structure Definitions], page 17, for more information.

structure function A structure function is a function created from a structure definition; the action of the structure function is defined by providing values to the properties in the input list, elaborating the structure, evaluating the properties in the output list, and providing those properties as the result of the function. See 7.8 [Structure Functions], page 88, for more information.

structure indicator A structure indicator is, given a component, the name following the double colon that indicates a structure with which the component will be associated. See 3.1.1 [Components], page 19, for more information.

structure A structure is the means by which a design is carved up into pieces by specifying components and their interconnection. See 3 [Structure], page 17, for more information.

subclass A class that names another class in its context is a subclass of the named class. See 4.1.3 [Type Classes], page 37, for more information.

succeed A pattern match succeeds if the pattern correctly describes the structure of the value against which it is being matched; a result of a successful pattern match is a binding from the names in the pattern to values in the subparts of the expression. See 6.3 [Patterns], page 72, for more information.

superclass A class that is named by another class in its context is a superclass of the class that names it. See 4.1.3 [Type Classes], page 37, for more information.

table A table is a tabular representation of information in the MHDL design which is given meaning by a generate definition. See 2.4 [Tables], page 13, for more information.

type classes A type class is a collection of name definitions (and possibly value definitions) for which values may be provided by a number of types, by providing an instance for each type that is to be a member of the type class. See 4.1 [Type Definitions], page 30, for more information.

type coercion Type coercion is the process by which an expression whose principal type does not agree with that required by the context in which the expression occurs is transformed to an expression whose principal type does agree with that required by context. See 4.3 [Type Coercions], page 51, for more information.

type specification A type specification is a statement of a type, which may be used in various parts of the description to establish the context in which expressions occur. See 4.1 [Type Definitions], page 30, for more information.

type definition A type definition is a binding between a type name and a type specification. See 4.1 [Type Definitions], page 30, for more information.

type A type is a set of values. Expressions belong to a type if the value which is represented by the expression is a member of the set of values of the type. See 4 [Types and Classes], page 29, for more information.

unit designation A unit designation is a specification of a dimension, along with an appropriate factor for that dimension, that gives the physical quantity of a value; for example, 'meters'. See 4.2.12.1 [Dimension Definition], page 44, for more information.

unit A unit definition binds a name to a unit designation or identifies the name as a basic unit for a dimension. See 4.2.12.2 [Unit Definition], page 47, for more information.

universal signals A universal signal is any signal, of whatever representation, for which normal operations (such as addition) are defined. See 10.4.1 [Universal Signals], page 163, for more information.

used structures A used structure is a structure that is indicated by a component in another structure. See 3.1.2 [Connectors], page 21, for more information.

value binding A value binding is a binding between a name and a value; each use of the name can be replaced by the value bound to that name without affecting the meaning of the design. See 6.2.2 [Pattern Bindings], page 69, for more information.

value definition A value definition is one or more pattern value pairs, where each pattern is matched against each value until one of the matches succeeds. This successful pattern match binds the names in the pattern to values. See 6 [Named Values], page 61, for more information.

value A value is a piece of information, such as an integer or floating point number, that gives information about an MHDL description. See 7 [Expressions], page 79, for more information.

variable A named value that is not a property. See 6 [Named Values], page 61, for more information.

visibility map A visibility map is a mapping between names and definitions. See 8.6 [Visibility], page 104, for more information.

well defined lambda list A well defined lambda list is a list of bound variables in an expression that is given explicitly in a lambda abstraction or by means of special names. See 7.9 [Derivatives and Integrals], page 89, for more information.

wildcard pattern A wildcard pattern is a pattern that always matches, and does not bind the matched value to any name. See 6.3 [Patterns], page 72, for more information.

Combined Index

!

!! 149

%

% 170
%% 155

&

&& 138

*

* 121
** 123

+

+ 121
++ 148

-

- 121

/

/ 123
/= 120

=

== 120

^

^* 170
^^ 170

<

< . . . 120
<= . . . 120
<> . . . 168

>

> . . . 120
>= . . . 120

—

—— . . . 138

A

abs . . . 121
absolute_configuration . . . 23
absReal . . . 170
abstract attribute . . . 55
abstract property . . . 55
abstract property definition . . . 55
abstract signal . . . 55
abstract_property_definition . . . 56
abstract_property_name_list . . . 56
accum . . . 155
accumArray . . . 155
acos . . . 123
acosh . . . 123
all . . . 149
alternative . . . 85
alternatives . . . 85
amap . . . 155
and . . . 149
any . . . 149
any_e . . . 110
any_sign . . . 110
apply to . . . 65
argument . . . 67, 81
arithmetic_sequence . . . 93
arity . . . 65
array . . . 155
Array . . . 155
arrays . . . 42
as_pattern . . . 71
ascii_escape . . . 112
asin . . . 123
asinh . . . 123

assign 155
assignment 57
Assoc 155
associated names 57
association_definition 58
associations 57
assocs 155
atan 123
atanh 123
attribute
    abstract 55
attribute_value 173
attributes 53

B

base 110
base_units 130
based literal 110
based_floating_point_part 110
based_integer_part 110
basic unit 47
basic_identifier 108
Bin 145
Binary 145
bind 148
binding 59
bodies 85
Bool 138
booleans 40
bot 79
bounds 155
break 149

C

case 85
case_configuration 23
case_expression 85
ceiling 124
Char 139
character_literal 112
characters 41
child_con_obj 175
child_conn_set 175
child_str_conn 175
child_str_obj 175

children 25
class
    function 168
    method 37
    operation 37
    type 29
class_body 37
class_constraint 31
class_definition 37
class_definition_list 37
class_header 37
class_list 36
class_placeholder 37
class_type_list 37
cmag 128
coercion
    type 51
column_header 13
column_name_list 13
comments 116
complex 43
Complex 128
complex numbers 43
complex_configuration 23
component configuration 22
    definitions 19
component indication 102
component_alternative 23
component_definition 20
component_list 17
components 19
composite types 148
concat 149
conditional_expression 85
configuration
    component 22
    definition 22
    structure 22
configuration_alternative 23
configuration_body 23
configuration_clause 23
configuration_definition 23
configurations 22
conjugate 128
connected_obj 174

connection list
    definition . . . 21
connection set . . . 21
connection_list . . . 21
connection_set . . . 174
connector
    definition . . . 21
connector types . . . 179
connector_clause . . . 17
connector_name_list . . . 17
connector_specification . . . 21
connector_type . . . 175
connectors . . . 21
constraint . . . 59
constraints . . . 75
constructed pattern . . . 71
constructor . . . 34
constructor_list . . . 31
Cont . . . 166
containing structure . . . 21
context . . . 31
continuous signals . . . 50
Continuous Signals . . . 166
ContPer . . . 167
control_character . . . 112
conversion_spec . . . 47
correct . . . 76
cos . . . 123
cosh . . . 123
cphase . . . 128
ctype . . . 175
currying . . . 67
cycle . . . 149

D

data tables . . . 115
decimal_floating_point_part . . . 110
decimal_integer_part . . . 110
decodeFloat . . . 124
default . . . 70
definite integral . . . 89
definition
    abstract property . . . 55
    default . . . 70
    dimension . . . 44

generate . . . 9
generated . . . 9
property value . . . 57
definition set . . . 9
standard . . . 117
table . . . 13
type . . . 29
unit . . . 47
definition_designator . . . 31
definition_list . . . 20
definitions
duplicate . . . 15
defint . . . 176
delay . . . 163
denominator . . . 170
deriv . . . 176
derivation . . . 6
derivation_clause . . . 31
derivative . . . 89
derivative_or_integral . . . 89
derived unit . . . 47
description unit . . . 15
design hierarchy . . . 25
children . . . 25
definition of . . . 17, 25
parents . . . 25
part . . . 25
root . . . 25
designation . . . 44
DFT . . . 167
differentiation . . . 89
digits . . . 110
dimension . . . 44
dimension definition . . . 44
dimension type . . . 44
dimension_definition . . . 44
Disc . . . 167
DiscPer . . . 167
discrete signals . . . 50, 167
div . . . 122
divRem . . . 122
drop . . . 149
dropWhile . . . 149

E

e . . . 123
elaboration . . . 26
electrical . . . 179
elem . . . 149
element type . . . 155
elems . . . 155
else_generate . . . 10
else_if_generate . . . 10
encodeFloat . . . 124
entity . . . 26
  creation of . . . 26
Enum . . . 122
enumerated types . . . 122
enumeration types . . . 37
enumFrom . . . 122
enumFromBy . . . 170
enumFromThen . . . 122
enumFromThenTo . . . 122
enumFromTo . . . 122
Eq . . . 120
equality . . . 120
equation . . . 59
equations . . . 76, 173
  satisfied . . . 76
error . . . 139
escaped_character . . . 112
escaped_identifier . . . 109
even . . . 122
Exception . . . 161
exceptions . . . 161
exists . . . 173
exp . . . 123
explicit_irrefutable_pattern . . . 71
exponent . . . 44, 110, 124
exposure_clause . . . 7
exposure_item . . . 7
exposure_list . . . 7
expression . . . 79
  case . . . 85
  if . . . 85
  typed . . . 87
expressions . . . 79
  associativity of . . . 79
  definition of . . . 79
  precedence of . . . 79

extended_digit . . . 110
extended_digits . . . 110
extract . . . 175

F

fetch . . . 155
field_list . . . 31
fields . . . 36
File . . . 145
filter . . . 149
Float . . . 127
floatApproxRational . . . 170
floatDigits . . . 124
floating_point_literal . . . 110
floatProperFraction . . . 170
floatRadix . . . 124
floatRange . . . 124
floor . . . 124
foldl . . . 149
foldl1 . . . 149
foldr . . . 149
foldr1 . . . 149
for_generate_definition . . . 10
for_generate_header . . . 10
format effector . . . 107
Fourier . . . 166
fractional . . . 123
Fractional . . . 123
freq . . . 167
fst . . . 170
full_name . . . 173
function . . . 59, 65, 67
Function . . . 168
function
  application of . . . 67
  argument type . . . 67
function binding . . . 63
function class . . . 168
  numeric . . . 176
  overloaded . . . 37
  partial application of . . . 67
  random . . . 177
  result type . . . 67
  standard . . . 168
  structural . . . 172

structure 87
function types 41
function_application 81
function_definition 37
function_designation 63
Functor 148

G

gap 112
gcd 170
general_alternative 85
general_pattern 71
generate definitions 9
generate_body 10
generate_definition 10
generate_list 10
generate_object_kind 10
generated definitions 9
generation scheme 94
generator 94
genericLength 149
graphic character 107
ground 182
Group 121
guard 85, 94
guard_pattern 85
guarded value definition 63
guarded_alternative 85
guarded_expression_list 63

H

head 149
hiding_clause 7
hiding_list 7
hierarchical names 102

I

identifier 108
if 85
if_configuration 23
if_generate_definition 10
if_generate_header 10
imagPart 128
implementation defined type 36
impulse 176

in 149
included unit 104
including unit 104
inclusion list 7
inclusion_list 7
indefinite integral 89
index 121
index types 121
indicated structure 20
indication
    component 102
indicator 44
indicators 44
indices 155
inheritance 6
inheritance_list 4
init 149
inRange 121
instance 29
    type 30
instance_body 39
instance_header 39
Int 125
integer_literal 110
integers 42
Integers 122
integral 89
integration 89
interface
    VHDL 177
InttoFloat 169
InttoRatio 169
inv_ztrans 167
invDFT 167
InvFourier 166
InvLaplace 166
irrefutable pattern 71
isAlpha 139
isAlphanum 139
isAscii 139
isControl 139
isDigit 139
isLower 139
isPrint 139
isSpace 139

isType 175
isUpper 139
iterate 149
Ix 121
ixmap 155

J

join 148

L

label 76
lambda_abstraction 83
Laplace 166
last 149
layouts 28
lazy evaluation 71
lazy matching 71
lazy matching of patterns 69
lcm 170
length 149
let_expression 84
letter_or_digit 108
lexical element 107
libraries 15
library_list 16
linear 65, 71
lines 139
list 93
List 149
list
  inclusion 7
list_pattern 71
listArray 155
lists 41
literal 91
LList 149
local_name 173
log 123
logBase 123

M

mag 167
map 148
max 120
maximum 149

maxInt . . . 170
method . . . 37
min . . . 120
minimum . . . 149
minInt . . . 170
mkPer . . . 167
mkPolar . . . 128
mkRectang . . . 128
mks_units . . . 130
mod . . . 122
model . . . 4
  definition of . . . 6
  derivations . . . 6
  inclusion list . . . 6
  standard . . . 177
model_body . . . 4
model_item . . . 4
Monad . . . 148
Monad0 . . . 148
MonadPlus . . . 148
Mult . . . 121
multiplier . . . 49

N

n_plus_k_pattern . . . 71
name . . . 99
  associated . . . 57
name definition . . . 59
  definition of . . . 97
  hierarchical . . . 102
  qualified . . . 101
  record . . . 100
  simple . . . 100
name_definition . . . 62
named value . . . 59
named_values . . . 62
names . . . 97
  mapping by configurations . . . 22
  special . . . 103
negate . . . 121
nested comments . . . 116
nested_definition . . . 84
nested_definitions . . . 84
newarray . . . 155
next_samp . . . 167

next_time 167
normal 161
not 138
notElem 149
nub 149
null 149
null type 42
null_expression 92
null_pattern 71
Num 121
numbers
    complex 43
numerator 170
numeric functions 176
numeric types 119

O

odd 122
offset_value 129
operation 37
operator 82
operator symbols 115
operator_application 82
or 149
ord 120
Ord 120
order
    total 120
ordinary comments 116
otherwise 138
overloaded 37

P

package 8, 9
    definition of 8
    standard 117
package_header 9
parenthesized_expression 92
parents 25
part 25
part creating structure 25
    creation of 26
partition 149
pattern 71
pattern binding 63, 69

default . . . 69
simple . . . 69
pattern matching . . . 59, 71
diverging . . . 71
failing . . . 71
succeeding . . . 71
Per . . . 167
periodic signal . . . 50
periodic signals . . . 167
phase . . . 167
Physical . . . 129
physical literal . . . 43
physical types . . . 43
physical_value . . . 95
pi . . . 123
Polar . . . 128
PolartoRectang . . . 169
polymorphic type . . . 30
polymorphic types . . . 29
polymorphic_name_list . . . 36
predefined types . . . 40
Primitive Types . . . 36
primitive_type_definition . . . 36
primitive_value . . . 96
principal type . . . 79
product . . . 149
products . . . 149
properFraction . . . 124
properties . . . 53
property
abstract . . . 55
creation of . . . 26
property value definitions . . . 57
property_kind . . . 56
property_name_list . . . 87
pshift . . . 167

Q

qualified name . . . 101

R

random functions . . . 177
range . . . 121
rangeSize . . . 170
Ratio . . . 126

Rational . . . 126
rationals . . . 42
RationaltoFloat . . . 169
readBin . . . 145
readFile . . . 145
readList . . . 141
ReadS . . . 141
readsPrec . . . 141
Real . . . 123
RealFloat . . . 124
RealFrac . . . 124
realPart . . . 128
reals . . . 43
recip . . . 123
record name . . . 100
record types . . . 36
RectangtoPolar . . . 169
reduce . . . 170
refutable pattern . . . 71
rem . . . 122
repeat . . . 149
reserved operators . . . 115
reserved word . . . 110
result . . . 148
reverse . . . 149
root, of a design hierarchy . . . 25
round . . . 124
row . . . 13

S

Scale . . . 129
scaled_value . . . 129
scaleFloat . . . 124
scan . . . 149
scanl . . . 149
scanl1 . . . 149
scanr . . . 149
scanr1 . . . 149
selected_name . . . 99
separator . . . 107
set
  definition . . . 9
showBin . . . 145
showFile . . . 145
showList . . . 141

ShowS 141
showsPrec 141
sigEq 163
sigGe 163
sigGt 163
sigLe 163
sigLt 163
signal
  abstract 55
  continuous 50, 166
  definition 50
  discrete 50, 167
  periodic 50, 167
signal types 163
  universal 163
signals 53
significand 124
signum 121
signumReal 170
simple names 100
simple_configuration 23
simple_expression 90
simple_name 99
simple_pattern 71
sin 123
single_character 112
sinh 123
snd 170
sort 149
span 149
special names 103
special_character 112
specified types 30
specified_type_definition 31
splitAt 149
sqrt 123
standard definitions 117
standard functions 168
standard models 177
StArray 155
str_electrical 179
strict 168
String 139
string_literal 112
strings 41

structural functions . . . 172
structure . . . 17
structure configuration . . . 22
  containing . . . 21
  definition of . . . 17
  definitions . . . 17
structure functions . . . 87
  input list . . . 87
  output list . . . 87
  indicated . . . 20
structure indicator . . . 20
  part creating . . . 25
  used . . . 21
structure_conn . . . 174
structure_definition . . . 17
structure_element . . . 17
structure_element_list . . . 17
structure_function_creation . . . 87
structure_obj . . . 174
sum . . . 149
summation . . . 167
sums . . . 149

T

table definition . . . 13
table_definition . . . 13
table_generate_definition . . . 10
table_generate_header . . . 10
table_header . . . 13
tables . . . 13
tail . . . 149
take . . . 149
takeWhile . . . 149
tan . . . 123
tanh . . . 123
text . . . 13
Text . . . 141
toLower . . . 139
toPhys . . . 169
Topology . . . 162
total orders . . . 120
toUpper . . . 139
transpose . . . 149
truncate . . . 124
tuple . . . 92

tuple_pattern . . . 71
tuples . . . 41
type classes . . . 29
type coercions . . . 51
  composite . . . 148
  connector . . . 179
type definition . . . 29
  definition of . . . 28
  dimension . . . 44
  element . . . 155
  enumerated . . . 122
  enumeration . . . 37
  function . . . 41
type header . . . 36
  implementation defined . . . 36
  index . . . 121
type instance . . . 30
  null . . . 42
  numeric . . . 119
  physical . . . 43
  polymorphic . . . 29, 30
  predefined . . . 40
  primitive . . . 36
  record . . . 36
  signal . . . 163
type specification . . . 30
  specified . . . 30
  standard . . . 117
type_component . . . 31
type_constructor . . . 31
type_definition . . . 36
type_list . . . 31
type_specification . . . 31
type_specifier . . . 31
typed_expression . . . 87
types . . . 28

U

underline . . . 108
unit . . . 47
  basic . . . 47
unit definition . . . 47
  derived . . . 47
  description . . . 15
  included . . . 104

including . . . 104
variations . . . 47
unit_conversion . . . 47
unit_definition . . . 47
unitof . . . 130
universal signals . . . 163
unlines . . . 139
until . . . 170
unwords . . . 139
update . . . 155
used structure . . . 21
Usig . . . 163

V

value binding . . . 69
value definition . . . 59
value_definition . . . 63
value_definition_target . . . 63
values . . . 28
variable . . . 59
variations . . . 47
variations_clause . . . 47
VHDL . . . 177
VHDL interface . . . 177
visibility . . . 104
visibility_item . . . 7

W

where_clause . . . 63
wildcard_pattern . . . 71
words . . . 139

Z

z_trans . . . 167
zero . . . 148
zip . . . 149
zip3 . . . 149
zip4 . . . 149
zip5 . . . 149
zip6 . . . 149
zip7 . . . 149
zipWith . . . 149
zipWith3 . . . 149
zipWith4 . . . 149
zipWith5 . . . 149

zipWith6 149
zipWith7 149