Oracle® Configurator

Constraint Definition Language Guide

Release 11*i*

Part No. B13606-03

May 2005

This book describes the Constraint Definition Language (CDL) semantics and syntax that pass validation by the Oracle Configurator parser and successfully compile when generating logic. CDL is used to define Statement Rules in Oracle Configurator Developer.

Oracle Configurator Constraint Definition Language Guide, Release 11*i*

Part No. B13606-03

Primary Authors: Tina Brand, Steve Damiani

Contributing Author: Harriet Shanzer

Contributors: Ivan Lazarov, Brian Ross

Contents

Send Us Your Comments ix

Preface xi

Intended Audience xi
Documentation Accessibility xi
Structure xii
Related Documents xii
Conventions xiii
Product Support xiii

1 Introduction

1.1 Overview of the Constraint Definition Language (CDL) 1-1
1.2 Relationships Expressed in CDL 1-1
1.3 Terminology 1-2
1.4 Syntax Notation 1-3

2 Principles of CDL

2.1 Before You Begin 2-1
2.1.1 What Model Structure Nodes and Properties Are Participants in the Rule? 2-1
2.1.2 Is the Model Structure Likely To Change Often? 2-1
2.1.3 What Does the Rule Need To Do? 2-1
2.1.4 What Types of Expressions Define the Relationships or Constraints You Need? 2-2
2.2 Anatomy of a Configuration Rule Written in CDL 2-2
2.2.1 Rule Definition 2-2
2.2.2 Rule Statements 2-3
2.2.3 Comments and Whitespace 2-3
2.2.4 Case Sensitivity 2-4
2.2.5 Quotation Marks 2-4
2.3 Data Types 2-4

3 Model Example

3.1 The House Model and its Window Submodel 3-1
3.2 Example Explicit Statements 3-2
3.3 Example Iterator Statements 3-3

3.4 CDL Flexibility 3-3
3.4.1 Incremental Rules 3-3
3.4.2 Alternative Rule Designs 3-4

4 CDL Elements

4.1 CDL Statements 4-1
4.1.1 Explicit Statements 4-1
4.1.2 Iterator Statements 4-2
4.1.2.1 Multiple Iterators in One Statement 4-2
4.2 Expressions 4-3
4.3 Keywords 4-4
4.3.1 CONSTRAIN 4-4
4.3.2 CONTRIBUTE...TO 4-5
4.3.2.1 CONTRIBUTE...TO with Decimal Operands and BOM Option Classes or Collections 4-5
4.3.3 COMPATIBLE...OF 4-5
4.3.4 FOR ALL....IN 4-6
4.3.5 WHERE 4-7
4.3.6 COLLECT 4-7
4.4 Operators 4-7
4.4.1 Predefined Operators Supported By CDL 4-7
4.4.2 Operator Results 4-9
4.4.3 Operator Precedence 4-9
4.4.4 LIKE and NOT LIKE Operators 4-10
4.4.5 Text Concatenation Operator 4-10
4.4.6 COLLECT Operator 4-10
4.5 Functions 4-12
4.5.1 Arithmetic 4-12
4.5.2 Trigonometric 4-13
4.5.3 Logical 4-14
4.5.4 Set 4-15
4.5.5 Text 4-15
4.5.6 Hierarchy or Compound 4-15
4.5.7 Function Overflows and Underflows 4-16
4.6 Operands 4-17
4.6.1 References 4-17
4.6.1.1 Model Object Identifiers 4-17
4.6.1.2 Simple Model Node References 4-18
4.6.1.3 Compound Model Node References Showing Context 4-18
4.6.1.4 Property References 4-19
4.6.2 Formal Parameters 4-20
4.6.2.1 Local Variables and Data Types 4-21
4.6.2.2 Local Variables and References 4-21
4.6.3 Literals 4-22
4.6.3.1 Numeric Literals 4-22
4.6.3.2 Boolean Literals 4-22
4.6.3.3 Text Literals 4-22

4.6.3.4 Collection Literals 4-23
4.7 Separators 4-24
4.8 Comments and Whitespace 4-25
4.8.1 Comments 4-25
4.8.2 Whitespace and Line Terminators 4-26

A CDL Formal Grammar

A.1 Notation Used in Presenting CDL Grammar A-1
A.1.1 Examples of Notation Used in Presenting CDL Grammar A-2
A.2 Terminal Symbols A-3
A.2.1 Keyword Symbols A-3
A.2.2 Operator Symbols A-4
A.2.3 Literal Symbols A-4
A.2.4 Separator Symbols A-5
A.2.5 Identifier Symbols A-5
A.2.6 Comment Symbols A-6
A.2.7 Whitespace Symbols A-7
A.3 Nonterminal Symbols A-7
A.4 EBNF Source Code Definitions for CDL Terminal Symbols A-9

B CDL Validation

B.1 Validation of CDL B-1
B.1.1 The Parser B-1
B.1.1.1 Calling the Oracle Configurator Parser B-1
B.1.1.2 The Parser's Validation Criteria B-1
B.1.2 The Compiler B-2
B.1.2.1 Calling the Oracle Configurator Compiler B-2
B.1.2.2 The Compiler's Validation Criteria B-2
B.2 The Input Stream to the Oracle Configurator Parser B-2
B.2.1 Unicode Characters B-3
B.3 Name Substitution B-3
B.3.1 Name Persistency B-3
B.3.2 Ambiguity Resolution B-3

Index

List of Examples

3–1 Example Explicit Statement in CDL 3-2
3–2 Example Iterator Statement in CDL 3-3
3–3 Incremental Rules and Their Equivalent As a Rolled Up Rule 3-3
3–4 Alternative Rule Designs With Equivalent Rule Intent 3-4
3–5 Alternative Rule Design with Narrowed Conditions 3-4
3–6 Alternative Rule Design using AllTrue function 3-4
4–1 Multiple Iterators in One CONSTRAIN Statement 4-2
4–2 Multiple Iterators in One CONTRIBUTE...TO Statement 4-3
4–3 Simple Mathematical Expression in a CDL Rule 4-3
4–4 Nested Mathematical Expression in a CDL Rule 4-3
4–5 Mathematical Expressions in Rule Statements 4-3
4–6 Expressions Resulting in a BOOLEAN Value 4-3
4–7 Expressions Resulting in an INTEGER or DECIMAL Value 4-4
4–8 Constraint Statements with the CONSTRAIN Keyword 4-4
4–9 Constraint Statements Without the CONSTRAIN Keyword 4-5
4–10 Constraint Statement with the FOR ALL...IN Iterator 4-5
4–11 CONTRIBUTE...TO Statements 4-5
4–12 Property-based Compatibility Rule 4-6
4–13 FOR ALL ... IN ... Clause 4-6
4–14 FOR ALL ... IN ... and WHERE Clause using Node Properties 4-6
4–15 LIKE Expression Resulting in a BOOLEAN Value 4-10
4–16 COLLECT Operator, Single Contribution 4-10
4–17 COLLECT Operator, Single Contribution 4-11
4–18 COLLECT Operator Contributions 4-11
4–19 COLLECT Operator with DISTINCT 4-11
4–20 Invalid Input Range Error 4-16
4–21 Intermediate Value Propagation Error 4-16
4–22 Calculated Input Value Out of Range Error 4-16
4–23 Calculated Value Not Within Valid Range Error 4-17
4–24 Full Path Model Node References 4-18
4–25 Relative Path Model Node Reference 4-19
4–26 Invalid Property References with CONTRIBUTE...TO Statements 4-20
4–27 Formal Parameter 4-21
4–28 Valid Local Variable of Inferred Data Type 4-21
4–29 Valid Formal Parameter and Reference 4-21
4–30 Formal Parameter and an Invalid Reference 4-21
4–31 Text Literals 4-22
4–32 Text Literal with Escapes 4-22
4–33 Multiple-Line Text Literal 4-23
4–34 A Valid Collection of Integer Literals 4-23
4–35 Valid Collection of Nodes 4-23
4–36 Valid Collections of Decimals 4-24
4–37 Invalid Collection 4-24
4–38 CDL Comments 4-25
4–39 Multiple Line Comments within a Statement Rule 4-25
A–1 EBNF for Keyword Symbols A-3
A–2 EBNF for Operator Symbols A-4
A–3 EBNF for Literal Symbols A-4
A–4 EBNF for Separator Symbols A-5
A–5 EBNF for Identifier Symbols A-6
A–6 EBNF for Comment Symbols A-6
A–7 EBNF for Whitespace Symbols A-7
A–8 EBNF for Nonterminal Symbols A-7
A–9 EBNF Source Code for Terminal Symbols A-9

List of Figures

3–1 Example House Model 3-2

List of Tables

1–1 Terminology Used in This Book 1-2
1–2 CDL Statement Syntax Notation........ 1-3
2–1 Kinds of Relationships or Constraints Available in CDL 2-2
2–2 Implicit Conversion of Data Type 2-4
4–1 CONTRIBUTE A TO B where B is a BOM Option Class or a Collection........ 4-5
4–2 Operators Listed by Type 4-8
4–3 Mapping of Operators and Data Types........ 4-9
4–4 Precedence of Operators 4-9
4–5 Arithmetic Functions........ 4-12
4–6 Trigonometric Functions........ 4-14
4–7 Logical Functions 4-14
4–8 Set Functions........ 4-15
4–9 Text Functions 4-15
4–10 Compound Function 4-15
4–11 Representations of Model Object Identifiers........ 4-17
4–12 Property References........ 4-19
4–13 Types of Numeric Literals 4-22
4–14 Escaped Characters Inside Double Quotes 4-23
4–15 Valid CDL Separators........ 4-25
A–1 Notation Used in Presenting CDL Grammar (EBNF) A-1
A–2 Values for Unicode Escapes Allowed in Identifiers A-6

Send Us Your Comments

Oracle Configurator Constraint Definition Language Guide, Release 11*i*

Part No. B13606-03

Oracle welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the title and part number of the documentation and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: czdoc_us@oracle.com
- FAX: 781-238-9898. Attn: Oracle Configurator Documentation
- Postal service:

  Oracle Corporation
  Oracle Configurator Documentation
  10 Van de Graaff Drive
  Burlington, MA 01803-5146
  USA

If you would like a reply, please give your name, address, telephone number, and electronic mail address (optional).

If you have problems with the software, please contact your local Oracle Support Services.

Preface

Welcome to the *Oracle Configurator Constraint Definition Language Guide*. This guide describes the semantics and syntax of the Constraint Definition Language or CDL. Use this document together with the other books in the Oracle Configurator documentation set to prepare for and implement rule definitions that are entered as text rather than created interactively in Oracle Configurator Developer. The text can be entered as a Statement Rule in Configurator Developer.

This preface describes how the guide is organized, who the intended audience is, and how to interpret the typographical conventions and syntax notation.

Intended Audience

This guide is intended for anyone responsible for creating and supporting rule definitions written in CDL, including Statement Rules in Oracle Configurator Developer. This guide assumes that you understand the kinds and behavior of configuration rules that are available in Oracle Configurator.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Structure

This guide contains a table of contents, lists of examples, tables and figures, a reader comment form, a preface, several chapters, appendixes, a glossary, and an index. Within the chapters, information is organized in numbered sections of several levels. Note that level does not imply importance or degree of detail. For instance, first-level sections in one chapter may not contain information of equivalent detail to the first-level sections in another chapter.

Chapter 1, "Introduction"

This chapter provides a high-level overview of CDL and the criteria for valid, executable rule definitions.

Chapter 2, "Principles of CDL"

This chapter introduces the principles of defining configuration rules using CDL.

Chapter 3, "Model Example"

This chapter introduces an example Model that is used to illustrate correct CDL semantics and syntax.

Chapter 4, "CDL Elements"

This chapter presents detailed information about the elements of CDL.

Appendix A, "CDL Formal Grammar"

This appendix provides a programmer's reference of CDL syntax.

Appendix B, "CDL Validation"

This appendix provides additional information about the Oracle Configurator parser's expectations and requirements during rule validation.

Glossary

This guide contains a glossary of terms used throughout the Oracle Configurator documentation set.

The Index provides an alternative method of searching for key concepts and product details.

Related Documents

For more information, see the following manuals in Release 11*i* of the Oracle Product documentation set:

- *Oracle Configurator Developer User's Guide*
- *Oracle Configurator Implementation Guide*
- *Oracle Configurator Installation Guide*
- *Oracle Configurator Extensions and Interface Object Developer's Guide*
- *Oracle Configurator Methodologies*

- *Oracle Configurator Modeling Guide*
- *Oracle Configurator Performance Guide*

Be sure you are familiar with the information and limitations described in the *About Oracle Configurator* documentation on Metalink, Oracle's technical support Web site.

Conventions

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

The following conventions are also used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example or relevant to the discussion have been omitted
boldface text	Boldface type in text indicates a new term, a term defined in the glossary, specific keys, and labels of user interface objects. Boldface type also indicates a menu, command, or option, especially within procedures
italics	Italic type in text, tables, or code examples indicates user-supplied text. Replace these placeholders with a specific value or string.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
>	The left bracket alone represents the MS DOS prompt.
$	The dollar sign represents the DIGITAL Command Language prompt in Windows and the Bourne shell prompt in Digital UNIX.
%	The percent sign alone represents the UNIX prompt.
`name()`	In text other than code examples, the names of programming language methods and functions are shown with trailing parentheses. The parentheses are always shown as empty. For the actual argument or parameter list, see the reference documentation. This convention is *not* used in code examples.

See Section 1.4, "Syntax Notation" for conventions specific to CDL.

Product Support

The mission of the Oracle Support Services organization is to help you resolve any issues or questions that you have regarding Oracle Configurator Developer and Oracle Configurator.

To report issues that are not mission-critical, submit a Technical Assistance Request (TAR) using Metalink, Oracle's technical support Web site at:

`http://www.oracle.com/support/metalink/`

Log into your MetaLink account and navigate to the Configurator TAR template:

1. Choose the **TARs** link in the left menu.
2. Click on **Create a TAR**.
3. Fill in or choose a profile.
4. In the same form:
   a. Choose **Product**: Oracle Configurator or Oracle Configurator Developer
   b. Choose **Type of Problem**: Oracle Configurator Generic Issue template
5. Provide the information requested in the iTAR template.

You can also find product-specific documentation and other useful information using MetaLink.

For a complete listing of available Oracle Support Services and phone numbers, see:

`www.oracle.com/support/`

Troubleshooting

Oracle Configurator Developer and Oracle Configurator use the standard Oracle Applications methods of logging to analyze and debug both development and runtime issues. These methods include setting various profile options and Java system properties to enable logging and specify the desired level of detail you want to record.

For general information about the logging options available when working in Configurator Developer, see the *Oracle Configurator Developer User's Guide*.

For details about the logging methods available in Configurator Developer and a runtime Oracle Configurator, see:

- The *Oracle Applications System Administrator's Guide* for descriptions of the Oracle Applications Manager UI screens that allow System Administrators to set up logging profiles, review Java system properties, search for log messages, and so on.
- The *Oracle Applications Supportability Guide*, which includes logging guidelines for both System Administrators and developers, and related topics.
- The Oracle Applications Framework Release 11i Documentation Road Map (Metalink Note # 275880.1).

1 Introduction

This chapter introduces the Constraint Definition Language and contains the following sections:

- Overview of the Constraint Definition Language (CDL)
- Relationships Expressed in CDL
- Terminology
- Syntax Notation

1.1 Overview of the Constraint Definition Language (CDL)

The Constraint Definition Language (CDL) is a modeling language. CDL allows you to define configuration rules, the constraining **relationship**s among items in configuration models, by entering them as text. A rule defined in CDL is an input string of characters that is stored in the CZ schema of the Oracle Applications database, validated by a **parser**, translated into executable code by a **compiler**, and interpreted at runtime by Oracle Configurator.

You use CDL to define a Statement Rule in Oracle Configurator Developer by entering the rule's definition as text rather than interactively assembling the rule's elements. Because you use CDL to define them, Statement Rules can express more complex constraining relationships than interactively defined configuration rules can.

See the *Oracle Configurator Developer User's Guide* for information about creating Statement Rules in Configurator Developer.

CDL also supports writing rules in rule-writing environments other than Configurator Developer for the purpose of importing rules directly into the CZ schema. For details about the availability of this functionality, see the latest *About Oracle Configurator* documentation on Metalink, Oracle's technical support Web site.

1.2 Relationships Expressed in CDL

Using CDL, you can define the following relationships that are supported by the rules available in Oracle Configurator Developer:

- Logical
- Numeric
- Property-based compatibility
- Comparison

The other types of relationships that can be defined in Configurator Developer (Explicit Compatibility rules and Design Charts) cannot be expressed in CDL.

For more information about the kinds of relationships that are supported in CDL, see Table 2–1, " Kinds of Relationships or Constraints Available in CDL" on page 2-2.

1.3 Terminology

Table 1–1 lists terms that are used throughout this guide. The Model that is used for all examples in this guide is explained in Chapter 3.

Table 1–1 Terminology Used in This Book

Term	Description
Cartesian product	A set of tuples that is constructed from two or more given sets and comprises all permutations of single elements from each set such that the first element of the tuple is from the first set and the second is from the second set, and so on.
clause	A segment of a rule statement consisting of a keyword and expression.
collection	A set of multiple operands within parentheses and separated by commas.
compiler	The part of Oracle Configurator that first parses rule definitions and then generates code that is executable at runtime.
explicit statement	Explicit statements express relations among explicitly identified participants and restrict execution of the rule to those participants and the Model containing those participants.
expression	A subset of the statement that contains operators and operands
formal identifier	A variable that is defined in the scope of an iterator statement to represent an iterating identifier.
iterator statement	Iterators are query-like statements that iterate, or repeat, over one or multiple relations or constraints.
non-terminal	The kind of symbols used in the notation for presenting CDL grammar that represent the names of grammar rules.
parser	A component of the Oracle Configurator compiler that analyzes the syntactic and semantic correctness of statements used in rule definitions.
relationship	A type of constraint expressed in a single statement or clause. A relationship can be equivalent to a simple rule. A Statement Rule expresses one or more relationship types but is not itself a type of relationship.
signature	The distinct combination of a function’s attributes, such as name, number of parameters, type of parameters, return type, mutability, and so on.
singleton	A single operand that is not within a collection.
statement	The entire sentence that expresses the rule’s intent. A CDL rule definition can consist of multiple statements, each consisting of clauses containing expressions, and separated by semi-colons.
terminal	The kind of symbols used in the notation for presenting CDL grammar that represent the names, characters, or literal strings of tokens.

Table 1–1 (Cont.) Terminology Used in This Book

Term	Description
token	The result of translating characters into recognizable lexical meaning. All text strings in the input stream to the parser, except whitespace characters and comments, are tokens. For more information about the use of special characters, see the *Oracle Configurator Developer User's Guide*.
unicode	A 16-bit character encoding scheme allowing characters from Western European, Eastern European, Cyrillic, Greek, Arabic, Hebrew, Chinese, Japanese, Korean, Thai, Urdu, Hindi and all other major world languages, to be encoded in a single character set.

See the Glossary for additional terms.

1.4 Syntax Notation

Table 1–2 describes the valid syntax notation for CDL. This notation is used throughout this book for CDL examples and in the syntax reference in Appendix A, "CDL Formal Grammar".

Table 1–2 CDL Statement Syntax Notation

Symbol	Description
-- or //	A double hyphen or double slash begins a single line comment that extends to the end of the line.
`/* */`	A slash asterisk and an asterisk slash delimits a multiline comment that can span multiple lines.
`&lower case`	Lower case prefixed by the ampersand sign is used for names of formal parameters and iterator local variables.
`UPPER CASE`	Upper case is used for keywords and names of predefined variables or formal parameters.
`Mixed Case`	Mixed case is used for names of user-defined Model nodes, names of user-defined rules
;	A semi-colon indicates the end of one statement and the beginning of the next

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Enter key at the end of a line of input.

See also Conventions in the Preface.

2 Principles of CDL

This chapter presents some principles to keep in mind when working in CDL, including the following topics:

- Before You Begin
- Anatomy of a Configuration Rule Written in CDL
- Data Types

2.1 Before You Begin

Before defining a rule in CDL, consider exploring the following key questions:

- What Model Structure Nodes and Properties Are Participants in the Rule?
- Is the Model Structure Likely To Change Often?
- What Does the Rule Need To Do?
- What Types of Expressions Define the Relationships or Constraints You Need?

2.1.1 What Model Structure Nodes and Properties Are Participants in the Rule?

The answer matters because it helps you choose which kind of CDL statement to use. CDL supports explicit and **iterator statement**s.You use **explicit statement**s to express relationships involving specifically named individual nodes or Properties in your Model structure. If you want a set of related nodes (such as all window models of a house) to participate in a series of identical rules, use iterator statements instead of repeating the same rule for each individual node.

2.1.2 Is the Model Structure Likely To Change Often?

If the structure is not static and expected to change often, you may want to define rules that use Properties, rather than explicitly including nodes in the rule's definition. This reduces the amount of required maintenance whenever the Model structure changes. For more information, see the *Oracle Configurator Modeling Guide*.

2.1.3 What Does the Rule Need To Do?

In other words, what type of relationship do you need to define? The answer matters because not all types of relationships can be expressed using CDL.

The available types of constraints and relationships that can be expressed in CDL include Logic, Numeric, Property-based Compatibility, and Comparison. See

Table 2–1, " Kinds of Relationships or Constraints Available in CDL" on page 2-2 for details.

For information about each type of relation, see the *Oracle Configurator Developer User's Guide*.

2.1.4 What Types of Expressions Define the Relationships or Constraints You Need?

Table 2–1 shows which CDL keywords are used to express which type of relationship. For example, to define a Numeric constraint that contributes a value of 10 to Total X when Option A is selected, use the CONTRIBUTE and TO keywords.

Table 2–1 Kinds of Relationships or Constraints Available in CDL

Rule Types	CDL Keywords
Logical or Comparison	Use the CONSTRAIN keyword and one operator. If you need to express a constraint between one or more options in your Model, then, at a minimum, use the CONSTRAIN keyword with the IMPLIES, EXCLUDES, DEFAULTS, NEGATES, or REQUIRES relation keyword
Numeric	Use the CONTRIBUTE and TO keywords when adding a value to a Numeric Feature, Option Count, Total, Resource, or the minimum or maximum total number of instances.
	Use the CONTRIBUTE (-1)* and TO keywords when subtracting a numeric values from a Numeric Feature, Option count, Total, Resource or instance count.
Compatibility	Use the COMPATIBLE keyword and at least two identifiers to indicate the nodes you want to compare.

2.2 Anatomy of a Configuration Rule Written in CDL

This section provides an overview of how the syntax, semantics, and lexical structure of a rule written in CDL relate to one another. This section contains the following topics:

- Rule Definition
- Rule Statements
- Comments and Whitespace
- Case Sensitivity
- Quotation Marks

For guidance in converting an existing rule to a Statement Rule, see the *Oracle Configurator Developer User's Guide*.

2.2.1 Rule Definition

A configuration rule has a name, associated Model, definition, other attributes such as Effectiveness and Usage, and optionally a description. The rule definition can be written in CDL and consists of whitespace characters, comments, and one or more individual statements that express the intent of the rule.

When creating a Statement Rule in Oracle Configurator Developer, you enter the name and description in input fields and the rule definition in the text box provided for that purpose.

For more information about entering rule definitions in Oracle Configurator Developer, see the *Oracle Configurator Developer User's Guide*.

2.2.2 Rule Statements

Statements define the rule's intent, such as to contribute a value of 10 to Total X when Option A is selected.

Multiple statements in a rule definition must be separated from one another with semi-colons (;). CDL supports two kinds of statements: Explicit and Iterator. For more information, see Section 4.1, "CDL Statements" on page 4-1.

CDL statements are parsed as **token**s; everything in CDL is a token, except whitespace characters and comments. For more information about how CDL is parsed, see Appendix B, "CDL Validation".

Statements consist of one or more **clause**s. Clauses consist of keywords and one or more **expression**s. Keywords are predefined tokens that determine CDL syntax and make it more readable and easy to use. CONSTRAIN and CONTRIBUTE are examples of keywords.

An expression is the part of a statement that contains an operator and the operands involved in a rule operation. An operator is a predefined keyword, function, or character that involves the operands in logical, functional, or mathematical operations. REQUIRES and the plus sign (+) are examples of operators. Operands are also called rule participants. An operand can be an expression, a literal, or an identifier. The literal or identifier operand can be present in the rule as a **singleton** or as a **collection**.

Literals are tokens of a specific data type, such as Numeric, Boolean (True or False), or Text. An identifier is a token that consists of a sequence of letters and digits. Identifiers identify Model objects or formal parameters. When an identifier identifies a Model object it refers to a Model node or Property and the sequence of letters and digits starts with a letter. These kinds of identifiers are called references. When an identifier is a formal parameter, it identifies a local variable and is used in an iterator statement. Formal parameters are a sequence of letters and digits prefixed with an ampersand (&).

For greater readability and to convey meaning such as the order of operations, CDL supports separators. Separators are tokens that maintain the structure of the rule by establishing boundaries between tokens, grouping them based on some syntactic criteria. Separators are single characters such as the semi-colon between statements or the parentheses around an expression.

For more information about these statements and the CDL elements they contain, see Chapter 4, "CDL Elements". For help with determining the CDL elements that correspond to particular rules, assemble a Logic, Numeric, Compatibility, or Comparison rule interactively in Oracle Configurator Developer, and then convert it to a Statement Rule. When you do this, Configurator Developer displays the rule's current definition in CDL. You can then expand or enhance the rule by typing additional statements, keywords, identifiers, structure node names, and so on.

2.2.3 Comments and Whitespace

Comments are included in rule definitions at your discretion to explain the rule. Whitespace, which includes spaces, line feeds, and carriage returns, format the input for better readability. See Section 4.8.1, "Comments" on page 4-25 and Section 4.8.2, "Whitespace and Line Terminators" on page 4-26 for details.

2.2.4 Case Sensitivity

Keywords are not case sensitive.

Keyword operators are not case sensitive.

Model object identifiers are case sensitive.

Formal parameters are case sensitive and cannot be in quotes.

The constants E and PI as well as the scientific E are not case sensitive.

The keywords TRUE and FALSE are not case sensitive.

Text literals are case sensitive.

All keywords, constant literals, and so on are not case sensitive.

Note: Operands are not case sensitive with the exception of Model object identifiers (node names), formal parameters or variables, User Property names, and text literals.

2.2.5 Quotation Marks

Model structure nodes with the same name as a keyword must be quoted when referred to in CDL

2.3 Data Types

Following are valid data types when defining a rule in CDL:

- INTEGER
- DECIMAL
- BOOLEAN
- TEXT
- Node types

Under certain circumstances, a data type of a variable is not compatible with the type expected as an argument. The Oracle Configurator parser does not support explicit conversion or casting between the data types. The parser performs implicit conversion between compatible types. See Table 2–2 for details.

If a rule definition has wrong data types, the parser returns a type mismatch error message. Example 4–37, "Invalid Collection" on page 4-24 shows a collection whose data types cannot be implicitly converted to be compatible.

Table 2–2 Implicit Conversion of Data Type

Source data type (or collection of the same type)	Implicitly converts to (or collection of the same type)
INTEGER	DECIMAL
NODE of type BOM Standard Item, BOM Option Class, BOM Model, Option Feature, Option, or Boolean Feature	BOOLEAN INTEGER DECIMAL Node type

Table 2–2 (Cont.) Implicit Conversion of Data Type

Source data type (or collection of the same type)	Implicitly converts to (or collection of the same type)
NODE of type Integer Feature	INTEGER
	DECIMAL
NODE of type Decimal Feature	DECIMAL
NODE of type Text Feature	TEXT

Unless specified otherwise, all references to matching types throughout this document assume the implicit data type conversions.

Note: Although TEXT is included as a data type here, it can only be used in a static context. You cannot use a TEXT literal, reference, or expression in the actual body of a CONSTRAINT or CONTRIBUTE expression. The Oracle Configurator compiler validates this condition when you generate logic for the Model.

3 Model Example

This chapter presents an example that is used throughout this book to illustrate CDL.

- The House Model and its Window Submodel
- Example Explicit Statements
- Example Iterator Statements
- CDL Flexibility

For help understanding the syntax notation of the examples in this and subsequent chapters, see Section 1.4, "Syntax Notation" on page 1-3.

3.1 The House Model and its Window Submodel

House is the parent Model, with Window its child Model.

Figure 3–1 Example House Model

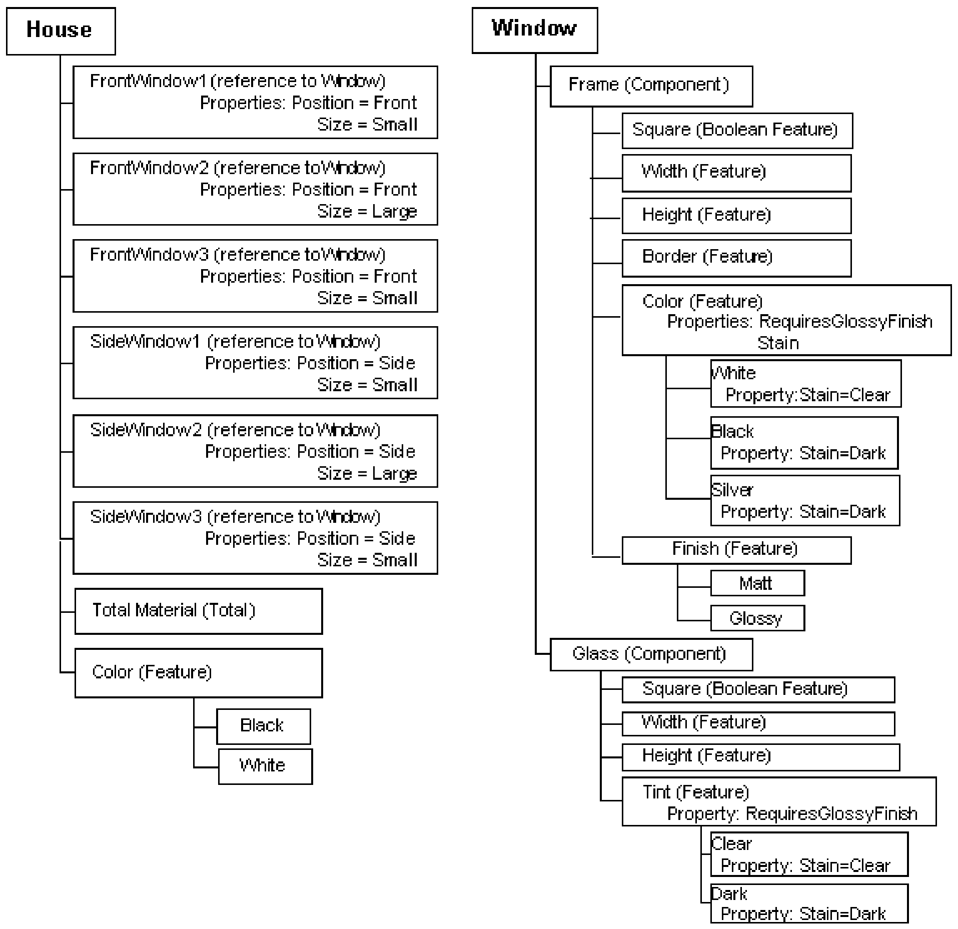

Figure 3–1 shows the Model `House` as the parent Model, with `Window` its child Model. The `Window` Model contains a `Frame` Component and a `Glass` Component with Features and Options.

3.2 Example Explicit Statements

For a description and general information about explicit statements, see Section 4.1.1 on page 4-1.

An example configuration rule calculates the size of glass to be put into a window frame for each Window instance. The glass is to be inserted into the Frame 1/2 inch at each side. To capture such a rule, you would provide a name, such as WindowGlassSize, a description, and then associate the rule with the `Window` Model.

Example 3–1 shows the definition of WindowGlassSize written in CDL.

Example 3–1 Example Explicit Statement in CDL

```
CONTRIBUTE Frame.Width - 2 * Frame.Border + 2 * 0.5
TO Glass.Width;
CONTRIBUTE Frame.Height - 2 * Frame.Border + 2 * 0.5
TO Glass.Height;
```

Two statements explicitly express two **Contributes to** relationships between the value of the Frame's dimensions and the glass to determine the required glass size. A semi-colon indicates the end of each statement and the whole rule definition.

3.3 Example Iterator Statements

For a description and general information about iterators, see Section 4.1.2 on page 4-2.

An example configuration rule constrains the window frame color so that for some colors, the finish is glossy. An iterator lets you define a rule that selects the glossy finish based on a Property.

In Example 3–2, the variable `&color` refers to all the Options of the Feature `Color`, in the `Frame` Component of the `Window` Model. The rule selects a glossy finish when one of those colors is selected AND the Property `RequiresGlossyFinish` is true.

Example 3–2 Example Iterator Statement in CDL

```
CONSTRAIN &color IMPLIES Frame.Finish.Glossy
FOR ALL &color IN OptionsOf(Frame.Color)
WHERE &color.Property ("RequiresGlossyFinish") = "True";
```

The reference to a particular Property value allows the constraining relation to be applied to a subset of the Color Options without explicitly naming the specific color. During validation, every node in `Color` is checked for a Property RequiresGlossyFinish. The result is that the rule iterates over all the children of `Color`, and for each color with the Property value set to `true`, the rule constrains the finish to `Glossy`.

The advantage of using an iterator statement is that if you add another color to the `Frame` Model, the rule definition does not have to be modified. Iterator statements significantly reduce the development and maintenance cost of the Model. With proper planning, the complete set of constraints could stay constant while the Model structure evolves over numerous publications.

For alternative rule definitions with similar intent, see Section 3.4, "CDL Flexibility" on page 3-3.

3.4 CDL Flexibility

CDL flexibly supports many ways of writing the same or similar rules. This section presents the following topics:

- Incremental Rules
- Alternative Rule Designs

3.4.1 Incremental Rules

CDL provides the flexibility to express complex rules as a series of incremental rules or those incremental rules as the subexpressions of a single, rolled up rule. Example 3–3 shows two rule anatomies that express the same behavior.

Example 3–3 Incremental Rules and Their Equivalent As a Rolled Up Rule

Incremental rules of a complex Numeric Rule.

```
CONTRIBUTE Frame.Width TO Glass.Width;
CONSUMES 2* Frame.Border FROM Glass.Width;
```

```
CONTRIBUTE 2 * 0.5 TO Glass.Width;
```

Rolled up complex Numeric Rule express the same behavior as the incremental rules.

```
CONTRIBUTE Frame.Width - (2 * Frame.Border) TO Glass.Width;
```

3.4.2 Alternative Rule Designs

As with Oracle Configurator, generally, CDL provides flexibility to express similar rule intent in various ways. Consider Example 3–2 on page 3-3, which could be designed differently, as shown in Example 3–4.

Example 3–4 Alternative Rule Designs With Equivalent Rule Intent

To select a glossy finish for every Option of the Feature `Color`, make the Boolean Property `RequiresGlossyFinish` imply a glossy finish.

```
CONSTRAIN &color.Property("RequiresGlossyFinish") IMPLIES Frame.Finish.Glossy
FOR ALL &color IN OptionsOf(Frame.Color)
```

In this rule, logic generation executes the rule on every `RequiresGlossyFinish` Property in the Options of the frame's `Color` Feature. Alternatively, you could write a WHERE clause that limits the rule to only those Options of the frame's `Color` Feature whose `RequiresGlossyFinish` Property equals true, as shown in Example 3–2.

Limiting the logic generation to the condition expressed in the WHERE clause is equivalent to applying a filter before execution, which usually results in better performance. When a rule iterates over a large number of options or combinations (for example, a Cartesian product), the WHERE clause does not necessarily improve performance.

Example 3–5 Alternative Rule Design with Narrowed Conditions

In Example 3–4, the rule binds `Frame.Finish.Glossy` to true at startup, merely because Property `RequiresGlossyFinish` exists. A different approach might be to add a `Special` Property that limits the Options over which the rule iterates to those that alone should have a glossy finish.

```
CONSTRAIN &color IMPLIES Frame.Finish.Glossy
FOR ALL &color IN OptionsOf(Frame.Color)
WHERE &color.Property ("Special") = "True"
AND &color.Property("RequiresGlossyFinish") = "True";
```

Here, the variable `&color` refers to all the Options of the Feature `Color`, in the `Frame` Component of the `Window` Model.   All Options in the Feature `Color` have the `Special` Property and this rule only iterates over those colors that are identified by `&color.Property("Special")= "True"`. Of that subset of colors, the rule selects a glossy finish when one of those colors is selected AND the Property `RequiresGlossyFinish` is true.

Example 3–6 shows the same rule intent as Example 3–5 using the AllTrue function.

Example 3–6 Alternative Rule Design using AllTrue function

```
CONSTRAIN AllTrue (&color, &color.Property ("RequiresGlossyFinish"))
IMPLIES Frame.Finish.Glossy
FOR ALL &color IN OptionsOf(Frame.Color)
WHERE &color.Property ("Special") = "True";
```

4 CDL Elements

Rules written in CDL include the following elements:

- CDL Statements
- Expressions
- Keywords
- Operators
- Functions
- Operands
- Separators
- Comments and Whitespace

For an overview of CDL elements, as well as details about case sensitivity and quotation marks, see Section 2.2, "Anatomy of a Configuration Rule Written in CDL" on page 2-2.

For syntax abstracts, see Appendix A, "CDL Formal Grammar"

4.1 CDL Statements

A rule definition written in CDL consists of one or more statements that define the rule's intent. The two kinds of statements are:

- Explicit Statements
- Iterator Statements

The difference between explicit and iterator statements is in the types of participants involved.

4.1.1 Explicit Statements

Explicit statements express relationships among explicitly identified participants and restrict execution of the rule to those participants and the Model containing those participants.

In an explicit statement, you must identify each node and Property that participates in the rule by specifying its location in the Model structure. An explicit statement applies to a specific Model, thus all participants of an explicit statement are explicitly stated in the rule definition.

CDL supports several kinds of explicit statements, which are identified by the keywords CONSTRAIN, CONTRIBUTE, and COMPATIBLE.

See Appendix A, "CDL Formal Grammar" for the syntax definition of statements.

Example 4–8 on page 4-4, shows such an explicit statement consisting of a single expression of the logical implies relation.

See Section 4.2, "Expressions" on page 4-3 for more information about the precise syntax of explicit statements.

4.1.2 Iterator Statements

Iterators are query-like statements that iterate, or repeat, over elements such as constants, Model references, or expressions of these. Iterators express relations among participants that are Model node elements of a collection or participants that are identified by their Properties and allow the rule to be applied to Options of Option Features with the same Properties. Iterators allow you to use the Properties of Model nodes to specify the participants of constraints or contributions. This is especially useful for maintaining persistent sets of constraints when the Model structure or its Properties change frequently. Iterators can also be used to express relationships between combinations of participants, such as with Property-based Compatibility Rules.

Iterator statements can use local variables that are bound to one or more iterators over collections. This is a way of expressing more than one constraint or contribution in a single implicit form. During compilation, a single iterator statement explodes into one or more constraints or contributions. See Section 4.4.6, "COLLECT Operator" on page 4-10 for more information.

The available iterators that make a rule statement an iterator statement are:

- FOR ALL....IN
- WHERE

See Appendix A, "CDL Formal Grammar" for the syntax definition of statements.

Example 4–10 on page 4-5 shows an iterator statement consisting of a single expression of the logical Defaults relation and the iterator.

See Section 4.2, "Expressions" on page 4-3 for more information about the precise syntax of kinds of iterator statements.

For an additional example of a rule statement that contains an iterator, see Example 3–2 on page 3-3.

4.1.2.1 Multiple Iterators in One Statement

The syntax of the FOR ALL clause allows for multiple iterators. The statement can be exploded to a **Cartesian product** of two or more collections.

Example 4–1, is an example of a Cartesian product as the rule iterates over all the Options of the `Tint` Feature in the `Glass` Component and over all the Options of the `Color` Feature in the `Frame` Component of the `Window` Model in Figure 3–1 on page 3-2. Whenever the `Stain` Property of the `Color` Options equals the `Stain` Property of the `Tint` Options, the selected color pushes the corresponding stain true. So, for example, when `&color.Property ("stain")` and `&tint.Property ("stain")` both equal Clear, selecting the White Option causes the Clear Option to be selected.

Example 4–1 Multiple Iterators in One CONSTRAIN Statement

```
CONSTRAIN &color IMPLIES &tint
FOR ALL
```

```
&color IN OptionsOf(Frame.Color),
&tint IN OptionsOf(Glass.Tint)
WHERE &color.Property ("stain") = &tint.Property ("stain");
```

The difference between this and a Property-based Compatibility Rule is that Example 4–1 *selects* participants without over constraining them, while a compatibility test *deselects* participants that do not pass the test. For more information on designing rules and the impact on performance, see Section 3.4.2, "Alternative Rule Designs" on page 3-4.

In Example 4–2, the numeric value of Feature `a` contributes to Feature `b` for all the Options of `a` and `b` when the value of their Property `Prop2` is equal.

Example 4–2 Multiple Iterators in One CONTRIBUTE...TO Statement

```
CONTRIBUTE &var1 TO &var2
FOR ALL &var1 IN {OptionsOf(a)}, &var2 IN {OptionsOf(b)}
WHERE &var1.Property("Prop2") = &var2.Property("Prop2");
```

4.2 Expressions

An expression is part of a CDL statement. It has two operands that are connected by an operator, or functions and their arguments. See Appendix A, "CDL Formal Grammar" for the syntax definition of expressions. See Section 4.4, "Operators", Section 4.6, "Operands", and Section 4.5, "Functions" for details.

Example 4–3 shows a simple mathematical expression where the two operands are `2` and `frame.border`, and the operator is `*` (multiplication).

Example 4–3 Simple Mathematical Expression in a CDL Rule

```
2 * frame.border
```

Example 4–4 shows a simple mathematical expression of Example 4–3 used as the second operand in another expression, where the first operand is `window.frame.width` and the operator is - (subtraction).

Example 4–4 Nested Mathematical Expression in a CDL Rule

```
window.frame.width - 2 * frame.border
```

See Section 4.4.3 on page 4-9 for details about the precedence of operators.

For an example of CDL rules using these expressions, consider the Window Model in Example 3–1. If you want to calculate the size of the glass to be put into a window frame where the glass is inserted in the frame 1/2 inch at each side, and the frame border is 1 inch, you might write the two Contributes To rules in Example 4–5.

Example 4–5 Mathematical Expressions in Rule Statements

```
CONTRIBUTE window.frame.width - 2 * frame.border + 2 * 0.5 TO glass.width;
CONTRIBUTE window.frame.height - 2 * frame.border + 2 * 0.5 TO glass.height;
```

Following are some additional examples of expressions.

Example 4–6 Expressions Resulting in a BOOLEAN Value

```
a > b
a AND b
```

```
(a + b) * c > 10
a.prop LIKE "%abc%"
```

Example 4–7 Expressions Resulting in an INTEGER or DECIMAL Value

```
a + b
((a + b) * c )^10
```

4.3 Keywords

Keywords consist of **Unicode** characters and are predefined identifiers within a statement.

Keywords include the following:

- CONSTRAIN
- CONTRIBUTE...TO
- COMPATIBLE...OF
- FOR ALL....IN
- WHERE
- COLLECT

See Section A.2.1, "Keyword Symbols" on page A-3 for the syntax definition of these keywords in expressions.

4.3.1 CONSTRAIN

The CONSTRAIN keyword is used at the beginning of a constraint statement. A constraint statement uses an expression to express constraining relationships. You can omit the CONSTRAIN keyword from a constraint statement.

Each constraint statement must contain one and only one of the following keyword operators:

- IMPLIES
- EXCLUDES
- REQUIRES
- NEGATES
- DEFAULTS

For a description of these constraints, see the section on Logic Rules in the *Oracle Configurator Developer User's Guide*.

Example 4–8 and Example 4–9 show constraint statements with and without the CONSTRAIN keyword.

Example 4–8 Constraint Statements with the CONSTRAIN Keyword

```
CONSTRAIN a IMPLIES b;
CONSTRAIN (a+b) * c > 10 NEGATES d;
```

Example 4–9 Constraint Statements Without the CONSTRAIN Keyword

```
a IMPLIES b;
(a + b) * c > 10 NEGATES d;
```

Example 4–10 expresses that if one Option of Feature `F1` is selected, then by default select all the rest of the Options. See Section 3.4.2 on page 3-4 for other examples of a CONSTRAIN statement with a FOR ALL iterator.

Example 4–10 Constraint Statement with the FOR ALL...IN Iterator

```
CONSTRAIN F1 DEFAULTS &var1
FOR ALL &var1 IN F1.Options();
```

4.3.2 CONTRIBUTE...TO

Unlike constraint statements, contribute statements contain numeric expressions. In a contribute statement, the CONTRIBUTE and TO keywords are required. See Section A, "CDL Formal Grammar" for the syntax definition of these keywords in expressions.

Example 4–11 CONTRIBUTE...TO Statements

```
CONTRIBUTE a TO b;
CONTRIBUTE (a + b) * c TO d;
```

CONTRIBUTE...TO is the CDL representation of the Numeric Rule in Oracle Configurator Developer.

For a description of a Contributes to rule, see the section on Numeric Rules in the *Oracle Configurator Developer User's Guide*.

4.3.2.1 CONTRIBUTE...TO with Decimal Operands and BOM Option Classes or Collections

Caution must be taken when writing rules with decimal operands and using BOM Option Classes, or collections. Table 4–1, explains what action should be taken when A contributes to B and B is either a BOM Option Class with multiple options, or B is a collection.

Table 4–1 CONTRIBUTE A TO B where B is a BOM Option Class or a Collection

If	AND	Then
A resolves to a decimal	Option 1 and Option 2 are both integers	Use the Round() function on A
	Option 1 and Option 2 are both decimals	No further action is needed on A
	Option 1 is decimal and Option 2 is integer	Use Round() function on A to meet the most limiting restriction - Option 2 an integer.
A in an integer	Option 1 and Option 2 are both integers	No further action is needed on A
	Option 1 and Option 2 are both decimals	
	Option 1 is decimal and Option 2 is integer	

4.3.3 COMPATIBLE...OF

The COMPATIBLE keyword is used at the beginning of a compatibility statement that defines compatibility based on Property values between Options of different Features, Standard Items of different BOM Option Classes, or between Options of a Feature and

Standard Items of a BOM Option Class. COMPATIBLE...OF is the CDL representation of a Property-based Compatibility Rule in Oracle Configurator Developer.

A Compatibility statement requires the keyword COMPATIBLE and two or more identifiers. The syntax of COMPATIBLE...OF is essentially the same as that of FOR ALL....IN. For each **formal identifier** in the COMPATIBLE clause, there must be a matching identifier in the OF clause. The conditional expression determining the set of desired combinations is in the WHERE clause.

The CDL of a Property-based Compatibility must include at least two iterators. For additional information about using a WHERE clause, see Section 4.3.5 on page 4-7.

In Example 4–12, the rule iterates over all the Options of the `Tint` Feature in the `Glass` Component and over all the Options of the `Color` Feature in the `Frame` Component of the `Window` Model in Figure 3–1 on page 3-2. A color and tint are compatible whenever the `Color` Option's `Stain` Property equals the `Tint` Option's `Stain` Property.

Example 4–12 Property-based Compatibility Rule

```
COMPATIBLE
&color OF Frame.Color,
&tint OF Glass.Tint
WHERE &color.Property("stain") = &tint.Property("stain");
```

For a description of Compatibility, including order of evaluation, see the section on Property-based Compatibility Rules in the *Oracle Configurator Developer User's Guide*.

See Section A, "CDL Formal Grammar" for the syntax definition of these keywords in expressions.

4.3.4 FOR ALL....IN

The FOR ALL and IN keywords begin the two clauses of an iterator statement. The IN keyword specifies the source of iteration

Note: The IN clause can contain only literal collections or collections of model nodes, such as OptionsOf. There is no specification of instances, so all instances of a given Model use the same iteration.

See Section A, "CDL Formal Grammar" for the syntax definition of these keywords in iterator expressions.

In Example 4–13, the result is 3 contributions to option `d`.

Example 4–13 FOR ALL ... IN ... Clause

```
CONTRIBUTE &var TO d
FOR ALL &var IN {a, b, c};
```

In Example 4–14, the result is as many contributions to Feature `d` as there are children in Feature `a`, whose Property `prop3` is less than 5. This example also shows a collection enclosed in braces (see Section 4.6.3.4, "Collection Literals" on page 4-23).

Example 4–14 FOR ALL ... IN ... and WHERE Clause using Node Properties

```
CONTRIBUTE &var.Property("NumProp") + 10 TO d
FOR ALL &var IN {OptionsOf(a)}
WHERE &var.Property("prop3") < 5;
```

In both examples, a single statement explodes into one or more constraints or contributions without explicitly repeating each one. In both examples, the iterator variable can also participate in the left hand side of the contribute statement.

4.3.5 WHERE

The WHERE keyword begins a clause of an iterator statement that acts as a filter to eliminate iterations that do not match with the WHERE criteria.

See Section A, "CDL Formal Grammar" for the syntax definition of this keyword in iterator expressions.

In Example 4–14, the result is only as many contributions to option `d` as there are children in the criteria specified in the WHERE clause.

Note: The conditional expression in the WHERE clause must be static. When using the COLLECT operation in a WHERE and an IN clause, the operands must be static. All User Properties on nodes and all constants are static operands. As a result, operands in the WHERE clause of a COMPATIBLE...OF statement can only be Properties.

Note: Configurator Developer evaluates Property-based Compatibility Rules from the top down, and gives no priority or precedence to an expression based on its use of the AND or OR operator. In other words, the system evaluates the first relation you enter, followed by the second, and so on.

4.3.6 COLLECT

The COLLECT keyword is used exclusively as an operator. For details about the COLLECT keyword, see Section 4.4.6, "COLLECT Operator" on page 4-10.

4.4 Operators

Operators are predefined tokens consisting of Unicode characters to be used as the expression operators among the expression operands. An operator specifies the operation to be performed at runtime between the operands. This section includes the following topics:

- Predefined Operators Supported By CDL
- Operator Results
- Operator Precedence
- LIKE and NOT LIKE Operators
- Text Concatenation Operator
- COLLECT Operator

4.4.1 Predefined Operators Supported By CDL

See Appendix A, "CDL Formal Grammar" for the syntax definition of operators. Table 4–2 lists the predefined operators supported by CDL.

Table 4–2 lists the predefined operators supported by CDL. Comparison types include comparing a numeric valued Feature with a property of a selected Option, or comparing a Property value with the name of an Option.

Table 4–2 Operators Listed by Type

Operator Type	Operators	Description
Logical	AND	AND requires two operands and returns true if both are true.
Logical	OR	OR requires two operands and returns true if either is true.
Logical	NOT	NOT requires one operand and returns its opposite value: false if the operand is true, true if the operand is false.
Logical	NotTrue	NotTrue requires one operand and returns true if its logic state is false or unavailable. For additional information about using NotTrue, see the *Oracle Configurator Modeling Guide*.
Logical	REQUIRES	REQUIRES requires two operands. See the *Oracle Configurator Developer User's Guide* for details.
Logical	IMPLIES	IMPLIES requires two operands. See the *Oracle Configurator Developer User's Guide* for details.
Logical	EXCLUDES	EXCLUDES requires two operands. See the *Oracle Configurator Developer User's Guide* for details.
Logical	NEGATES	NEGATES requires two operands. See the *Oracle Configurator Developer User's Guide* for details.
Logical	DEFAULTS	DEFAULTS requires two operands. See the *Oracle Configurator Developer User's Guide* for details.
Logical and Comparison	LIKE	LIKE requires two text literal operands and returns true if they match. See Section 4.4.4 for restrictions.
Logical and Comparison	NOT LIKE	NOT LIKE requires two text literal operands and returns true if they do not match. See Section 4.4.4 for restrictions
Logical, Arithmetic, and Comparison	=	Equals requires two operands and returns true if both are the same.
Logical, Arithmetic, and Comparison	>	Greater than requires two operands and returns true if the first is greater than the second.
Logical, Arithmetic, and Comparison	<	Less than requires two operands and returns true if the first is less than the second.
Logical, Arithmetic, and Comparison	<>	Not equal requires two operands and returns true if they are different.
Logical, Arithmetic, and Comparison	<=	Less than or equal to requires two operands and returns "true" if the first operand is less than or equal to the second.
Logical, Arithmetic, and Comparison	>=	Greater than or equal requires two operands and returns "true" if the first operand is greater than or equal to the second.

Table 4–2 (Cont.) Operators Listed by Type

Operator Type	Operators	Description
Arithmetic	*	Performs arithmetic multiplication on numeric operands.
Arithmetic	/	Performs arithmetic division on numeric operands.
Arithmetic	-	Performs arithmetic subtraction on numeric operands.
Arithmetic	+	Performs arithmetic addition on numeric operands.
Arithmetic	^	Performs arithmetic exponential on numeric operands.
Arithmetic	%	Performs arithmetic modulo on numeric operands.
Text	+	Performs a concatenation of text strings. See Section 4.4.5 for restrictions.
Other	( ) , . -	parentheses ( ) are used to group sub-expressions comma (,) is used to separate function arguments dot (.) is used for referencing objects in the Model tree structure unary minus (-) is used to make positive values negative and negative values positive.

4.4.2 Operator Results

The result of each expression operator can participate as an operand of another operator as long as the return type of the former matches with the argument type of the latter. See Section 2.3, "Data Types" on page 2-4 for more information about allowable data types of operands.

Table 4–3 lists the basic return types of the operators.

Table 4–3 Mapping of Operators and Data Types

Operator(s)	Data type
Arithmetic	INTEGER, DECIMAL
Logical	BOOLEAN
Comparison	BOOLEAN

4.4.3 Operator Precedence

Operators are processed in the order given in the following list. Operators with equal precedence are evaluated left to right.

Table 4–4 lists the precedence of expression operators in CDL.

Table 4–4 Precedence of Operators

Operator	Precedence (direction)	Description
()	1 (right)	Parenthesis
.	2 (left)	Navigation
^	3 (right)	Arithmetic power
Unary `+, -` `NOT, NotTrue`	4	Unary plus and minus, Not and NotTrue
`*, /, %`	5 (left)	Arithmetic multiplication and divisions

Table 4–4 (Cont.) Precedence of Operators

Operator	Precedence (direction)	Description
Binary `+, -`	6 (left)	Arithmetic plus and minus, text concatenation
`<, >, =, <=, >=, <>`	7 (left)	Comparison operators
`LIKE, NOT LIKE`		
`AND`	8 (left)	Logical AND
`OR`	9 (left)	Logical OR
`DEFAULTS, EXCLUDES, NEGATES, IMPLIES, REQUIRES`	10 (left)	Logic operators

4.4.4 LIKE and NOT LIKE Operators

Although LIKE and NOT LIKE are included as text relational operators, they can only be used in static context; for example, the WHERE clause of iterators. As with any TEXT data type, you cannot use LIKE and NOT LIKE with runtime participants unless it evaluates to a constant string. Oracle Configurator Developer validates this condition when you generate logic.

Example 4–15 LIKE Expression Resulting in a BOOLEAN Value

```
a.prop LIKE "%eig%"
```

A `TRUE` result is returned if the text of `a.prop` contains the characters ’eig’, such as a.prop =’weight’ or ’eight’. `FALSE` is returned if the text of a.prop=’rein’. For more information on the LIKE operator and the use of wildcards, see the *Oracle Configurator Developer User’s Guide*.

For a list of comparison operators, see Table 4–2 on page 4-8.

4.4.5 Text Concatenation Operator

Although "+" is included as a text concatenation operator, it can only be used in static context; for example, the WHERE clause of iterators. As with any TEXT data type, you cannot use text concatenation in the actual body of a constrain or contributor statement unless it evaluates to a constant string. Oracle Configurator Developer validates this condition when you generate logic.

4.4.6 COLLECT Operator

A collection of values can be created using an aggregation function such as Min(...), Max(...), Sum(...), AnyTrue(...). An iterator can use the COLLECT operator to specify the domain of the collection that is passed to the aggregation function. In many cases FOR ALL serves that purpose. Example 4–16 shows a single contribution of the maximum value of the collection of children of Feature `a` using a COLLECT operator and a FOR ALL iterator.

Example 4–16 COLLECT Operator, Single Contribution

```
CONTRIBUTE Max(COLLECT &var FOR ALL &var IN {OptionsOf(a)}) TO d;
```

has the same result as

```
CONTRIBUTE Max &var TO d
```

```
FOR ALL &var IN {OptionsOf(a)} ;
```

The COLLECT operator is necessary when limiting an aggregate. Example 4–17 shows a rule where the iteration of the FOR ALL and WHERE clauses result in an error for every element of the collection `{Option11, Option32, OptionsOf(Feature1)}` that does not contain the Property `P1`.

Example 4–17 COLLECT Operator, Single Contribution

```
CONSTRAIN &varA IMPLIES Component.Featuure.Option
FOR ALL &varA IN {Option11, Option32, OptionsOf(Feature1)}
WHERE &varA.Property("P1") = 5;
```

Example 4–18 uses COLLECT, which prevents the error.

Example 4–18 COLLECT Operator Contributions

```
CONSTRAIN &varA IMPLIES Component.Featuure.Option
FOR ALL &varA IN {Option11, Option32, {COLLECT &varB
                                       FOR ALL &varB IN OptionsOf(Feature2)
                                       WHERE &varB.Property("P1") = 5}};
```

COLLECT can be used in any context that expects a collection. The COLLECT operator can be used along with a complex expression and a WHERE clause for filtering out elements of the source domain of the collection. See Section 4.3.5, "WHERE" on page 4-7 for more information.

Since COLLECT is an operator that returns a collection, it can also be used inside of a collection literal, as long as the collection literal has a valid inferred data type. The Oracle Configurator compiler flattens the collection literal during logic generation, which allows collections to be concatenated. See Section 4.6.3.4, "Collection Literals" on page 4-23 for details.

The COLLECT operator can have only one iterator, because the return type is a collection of singletons. CDL does not support using a Cartesian product with the COLLECT operator.

The COLLECT operator cannot put dynamic variables in the IN and WHERE clauses, as this may result in a collection that is unknown at compile time. For additional information, see Section 4.3.5, "WHERE" on page 4-7.

The COLLECT operator can use the DISTINCT keyword to collect distinct values from a Property, as shown in Example 4–19, which prevents the selection of options having different values for the Property `Shape` from the Option Feature `Feature3`. `Feature3` has zero Minimum Selections and no limit on Maximum Selections.

Example 4–19 COLLECT Operator with DISTINCT

```
AnyTrue({COLLECT &opt1
        FOR ALL &opt1 IN {'Feature3'.Options()}
        WHERE &opt1.Property("Shape") = &shape})
EXCLUDES
AnyTrue({COLLECT &opt2
        FOR ALL &opt2 IN {'Feature3'.Options()}
        WHERE &opt2.Property("Shape") <> &shape})
FOR ALL &shape IN
        {COLLECT DISTINCT &node.Property("Shape")
        FOR ALL &node IN 'Feature3'.Options()}
```

4.5 Functions

In addition to operators, expressions can also contain functions, which may take arguments and return results that can be used in the rest of the expression. All standard mathematical functions are implemented in CDL.

The result of each function can participate as an operand of another operator or function as long as the return type of the former matches with the argument type of the latter.

Functions perform operations on their arguments and return values which are used in evaluating the entire statement. Functions must have their arguments enclosed in parentheses and separated by commas if there is more than one argument. Function arguments can be expressions.

For example, both of the following operations have the correct syntax for the Round function, provided that Feature-1 and Feature-2 are numeric Features:

```
Round (13.4)
Round (Feature-1 / Feature-2)
```

CDL supports the following functions:

- Arithmetic
- Trigonometric
- Logical
- Set
- Text
- Hierarchy or Compound

This section also contains information about Function Overflows and Underflows on page 4-16

4.5.1 Arithmetic

Table 4–5 lists the arithmetic functions that are available in CDL. The term infinity is defined as a number without bounds. It can be either positive or negative.

Table 4–5 Arithmetic Functions

Function	Description
Abs(x)	Takes a single number as an argument and returns the positive value (0 to +infinity). The domain range is -infinity to +infinity. Returns the positive value of x. Abs(-12345.6) results in 12345.6
Round(x)	Takes a single decimal number as an argument and returns the nearest integer. If the A side of a numeric rule is a decimal number, contributing to an imported BOM that accepts decimal quantities, then the Round(x) function is unavailable. The reason that the Round(x) function is unavailable is that the contributed value does not need to be rounded as the B side accepts decimal quantities. This function is available when the BOM item accepts only integer values.
RoundDownToNearest(x,y)	This is a binary function. x is a number between -infinity and +infinity, y is a number greater than 0 and less than +infinity. A number is returned between -infinity and +infinity. The first argument is rounded to the nearest smaller multiple of the second argument. For example, RoundDownToNearest(433,75) returns 375.

Table 4–5 (Cont.) Arithmetic Functions

Function	Description
RoundToNearest(x,y)	This is a binary function. x is a number between -infinity and +infinity, y is a number greater than 0 and less than +infinity. A number is returned between -infinity and +infinity. RoundToNearest(433,10) returns 430.
RoundUpToNearest(x,y)	This is a binary function. The number x is between -infinity and +infinity, and the number y is greater than 0 and less than +infinity. A number is returned between -infinity and +infinity. The first argument is rounded up to the nearest multiple of the second argument. For example, RoundUpToNearest(34.1,0.125) returns 34.125.
Ceiling(x)	Takes a single decimal number as an argument and returns the next higher integer. For example, ceiling(4.3) returns 5, and ceiling(-4.3) returns -4.
Floor(x)	Takes a single decimal number as an argument and returns the next lower integer. For example, floor(4.3) returns 4, and floor(-4.3) returns -5.
Log(x)	Takes a single number greater than 0 and less than +infinity and returns a number between -infinity and +infinity. Returns the logarithmic value of x. An error occurs if x=0.
Log10(x)	Takes a single number greater than 0 and less than +infinity and returns a number between -infinity and +infinity. Returns the base 10 logarithm of x. An error occurs if x=0.
Min(x,y,z...)	Returns the smallest of its numeric arguments.
Max(x,y,z...)	Returns the largest of its numeric arguments.
Mod(x,y)	This is a binary function. Returns the remainder of x/y where x and y are numbers between -infinity and +infinity. If y is 0, then division by 0 is treated as an error. If x=y, then the result is 0. For example, Mod(7,5) returns 2.
Exp(x)	Returns e raised to the x power. Takes a single number between -infinity and +infinity and returns a value between 0 and +infinity.
Pow(x,y)	This is a binary function. Returns the result of x raised to the power of y. The number x is between -infinity and +infinity. The integer y is between -infinity and +infinity and the returned result is between -infinity and +infinity. If y=0, then the result is 1. For example, Pow(6,2) returns 36.
Sqrt(x)	Sqrt(x) returns the square root of x. Takes a single number between 0 and +infinity and returns a value between 0 and +infinity. An input of -x results in an error.
Truncate(x,y)	Truncate(x,y) takes a single number x and truncates it to the number of y integers after the decimal point. The default value of y is 0. For example, truncate(4.15678) returns 4 and truncate(4.15678,2) returns 4.15.

4.5.2 Trigonometric

Table 4–6 lists the compound functions that are available in CDL

Table 4–6 Trigonometric Functions

Function	Description
Sin(x)	Takes a single number x between -infinity and +infinity and returns a value between -1 and +1.
ASin(x)	Takes a single number between -1 and +1 and returns a value between -pi/2 and +pi/2. ASin(x) returns the arc sine of x. An input outside the range between -1 and +1 results in an error.
Sinh(x)	Returns the hyperbolic sine of x in radians. Takes a single number between -infinity and +infinity and returns a value between -1 and +infinity. An error is returned when the result exceeds the double. For example, sinh(-99) is valid but sinh(999) results in an error.
Cos(x)	Takes a single number between -infinity and +infinity and returns a value between -1 and +1. Returns the cosine of x.
ACos(x)	Takes a single number between -1 and +1 and returns a value between 0 and pi. ACos(x) returns the arc cosine of x. An input outside the range between -1 and +1 results in an error.
Cosh(x)	Takes a single number between -infinity and +infinity and returns a value between -infinity and +infinity. Returns the hyperbolic cosine of x in radians. An error is returned if x exceeds the max of a double: cosh(-200) is valid whereas cosh(-2000) results in an error.
Tan(x)	Takes a single number x between -infinity and +infinity and returns a value between -infinity and +infinity.
ATan(x)	Takes a single number between -infinity and +infinity and returns a value between -pi/2 and +pi/2. ATan(x) returns the arc tangent of x.
Tanh(x)	Returns the hyperbolic tangent of x. Takes a single number x between -infinity and +infinity and returns a value between -1 and +1.
ATan2(x,y)	The arc tangent function is a binary function. The x and y values are between -infinity and +infinity. It returns a value between -pi and +pi. This is the four-quadrant tangent inverse.

4.5.3 Logical

Table 4–7 lists the logical functions that are available in CDL.

Table 4–7 Logical Functions

Function	Description
AllTrue	A logical AND expression. Accepts one or more logical values or expressions. Returns true if all of the arguments are true, or false if any argument is false. Otherwise, the value of AllTrue is unknown.
AnyTrue	A logical OR expression. Accepts one or more logical values or expressions. Returns true if any of the arguments are true, or false if all arguments are false. Otherwise, the value of AnyTrue is unknown.
NotTrue	Accepts a single logical value or expression. Returns True if the argument is False or unknown. If the argument is True, the value of NotTrue is unknown. For additional information about using NotTrue, see the *Oracle Configurator Modeling Guide*.

4.5.4 Set

Table 4–8 lists the set functions that are available in CDL.

Table 4–8 Set Functions

Function	Description
Count	Returns the count or number of members in the collection.
Min	Returns the smallest numeric member in the collection.
Max	Returns the largest numeric member in the collection.

4.5.5 Text

Although the Text functions are included here, they can only be used in static context; for example the WHERE clause of iterators.

> **Note:** As with any TEXT data type, do not use a text function in the body of a CONSTRAINT or CONTRIBUTE statement unless it evaluates to a constant string. The compiler validates this condition.

Table 4–9 lists the text functions that are available in CDL

Table 4–9 Text Functions

Function	Description
Matches	Compares two operands of text literals and returns true if they match.
NotMatches	Compares two operands of text literals and returns true if they do not match.
BeginsWith	Compares two operands of text literals and returns true if the first begins with the character(s) of the second.
EndsWith	Compares two operands of text literals and returns true if the first ends with the character(s) of the second.
Equals	Compares two operands of text literals and returns true if the first equals the second.
NotEquals	Compares two operands of text literals and returns true if the first does not equal the second

4.5.6 Hierarchy or Compound

In addition, several functions are available to support backward compatibility for functions in Configurator Developer that operate over the Model structure hierarchy.

Table 4–10 lists the compound function that is available in CDL.

Table 4–10 Compound Function

Function	Description
OptionsOf	Takes BOM Option Class, Component, or Feature as an argument and returns its Options.

4.5.7 Function Overflows and Underflows

It is possible that some arithmetic functions produce an error either because of the resulting size (larger than the largest positive or negative double) or an invalid input. Entering a meaningful rule violation message can be helpful when debugging errors.

For more information about violation messages, see the *Oracle Configurator Developer User's Guide*.

Following are some examples of possible error messages.

Example 4–20 Invalid Input Range Error

Consider a Numeric rule in which Acos(A-integer) contributes to a Total. When the input is out of the valid domain range (-1 to 1), Oracle Configurator returns the following error message.

```
There is a contradiction selecting A-Integer
```

To enhance the usability of this error based on the particular rule, you can specify the following violation message to appear after the system error.

```
Calculation of ACos(x) - x is not a valid value.
```

Example 4–21 Intermediate Value Propagation Error

It is possible that propagation through some math functions results in an unexpected error because of an intermediate value propagated to the argument of the function. The following Model has a Feature with two counted Options (Option1 and Option2), a Resource (R) with no initial value (default is 0), and a Total (T) with no initial value (default is 0).

```
Numeric Rule 1: Contribute Option1 *-1 to R
```

```
Numeric Rule 2: Sqrt(R) contribute to T
```

If `Option1` is 1, then `R` has a value of -1. Numeric rule 2 tried to calculate the Sqrt(-1) and Oracle Configurator returns the following error message.

```
There is a contradiction selecting Option1.
```

To enhance the usability of this error based on the particular rule, you can specify the following violation message to appear after the system error.

```
The result of an intermediate rule gives R an invalid value.
```

Example 4–22 Calculated Input Value Out of Range Error

The following Model has a Boolean Feature (B1), a Feature with two counted Options (Option1 and Option2), a Resource (R) with no initial value (default is 0), and a Total (T) with no initial value (default is 0).

```
Numeric Rule 1: (Option1)*2000 contribute to T
```

```
Numeric Rule 2: Contribute CosH(T) * -1 to R
```

If `Option1` is 1, then `T` has a value of 2000 and `CosH(T)` produces a result that is greater than the max of a double and Oracle Configurator returns the following error message.

```
There is a contradiction selecting Option1.
```

To enhance the usability of this error based on the particular rule, you can specify the following violation message to appear after the system error.

```
This rule uses the value calculated from Numeric Rule 1 - Option1 * 2000.
```

Example 4–23 Calculated Value Not Within Valid Range Error

A Model has two integer Features (I1 and I2) with initial values of 0. Totals (T1 and T2) with no specified initial values. The Numeric rule ACos(I1-I2) contributes to T1. If I1 is 1 and I2 is 3, then I1-I2 is outside the valid range (-1 to 1) for ACos(x). Oracle Configurator returns the following error message.

```
There is a contradiction selecting I2.
```

To enhance the usability of this error based on the particular rule, you can specify the following violation message to appear after the system error.

```
The resulting value of I1 - I2 is outside the valid range of ACos(x).
```

Note: This behavior depends on the order in which the relations are propagated.

4.6 Operands

Operands are the rule participants upon which the actions of keywords and operators are executed. The following are kinds of operands:

- References
- Formal Parameters
- Literals

See Appendix A, "CDL Formal Grammar" for the syntax definition of operands.

4.6.1 References

References are identifiers that refer to Model objects by name (Model nodes or Model Properties). At runtime, the object or its value is used in the rule. A reference could be a Model node or a Property reference.

4.6.1.1 Model Object Identifiers

A Model object identifier is a token that refers by name to a particular object in the Model structure. At runtime it's the node or the value that is actually used in the rules they participate in depending on the context.

Model object identifiers have two different representations: quoted with the single quote ('...'), or not quoted. They do not have to be quoted if they refer to Model nodes with names that contain only letters or digits, but they must be quoted otherwise (for example, if the name contains a space, special character or is the same as a keyword.).

For example:

Table 4–11 Representations of Model Object Identifiers

Model Object Identifier	Refers to...
`House`	Model called House.

Table 4–11 (Cont.) Representations of Model Object Identifiers

Model Object Identifier	Refers to...
`'House'`	Model called House.
`'Total Material'`	Node called Total Material. Because the node name contains a space, it must be quoted.

See Section 4.6.1, "References" on page 4-17 for details.

4.6.1.2 Simple Model Node References

References to Model nodes can be made through simple identifiers specifying the name of the Model node. Model node references are context dependent. Since there is no restriction on uniqueness of the Model node names, just using a Model identifier may be ambiguous depending on the context.

For example, based on the House Model shown in Figure 3–1, "Example House Model" on page 3-2, the context of the property Color is ambiguous because it could refer to the Frame or to the House.

```
Color -- this refers to Color in Frame
```

But in the context of House:

```
Color -- this refers to Color in House
```

Descendant Model nodes in the current Model context are always with higher priority than ancestor nodes. Thus Color in context Frame is not ambiguous (since Frame is a descendant node of House), but Color of the House is. When it is not possible to uniquely refer to a model node in the context of a rule, the rule developer must use compound identifiers (see Section 4.6.1.3).

4.6.1.3 Compound Model Node References Showing Context

Compound Model node references are sequences of Model node identifiers separated by the dot character (.). Compound references uniquely identify Model nodes in a particular context by presenting Model node paths. Compound references are necessary because Configurator Developer allows Model nodes to have identical names in the same Model structure as long as they are not siblings. In other words, Compound references are used for navigation in the Model structure.

The most explicit path is the full path. A full path contains all levels of the hierarchy by node name, including Model, nested components, references, Option Classes, and Features.

In Example 4–24, the first line shows the full path of Option `Dark` in the Feature `Tint` in the Component `Glass` in the referenced Model SideWindow2 in the parent Model House. The second line shows the full path of Color in Frame in FrontWindow1 in House.

Example 4–24 Full Path Model Node References

```
...
House.SideWindow2.Glass.Tint.Dark
...
House.FrontWindow1.Frame.Color
...
```

You can omit any head of the path that does not disambiguate the reference. So to refer precisely and only to `Color` in the context of `House`, you must specify enough of the head of the path. The reference in Example 4–25 unambiguously refers to `Color` in `Frame` in `FrontWindow1` in the `House` Model.

Example 4–25 Relative Path Model Node Reference

```
...
FrontWindow1.Frame.Color
...
```

4.6.1.4 Property References

Identifiers that refer to System and User Properties of Model nodes must also be compound. When referring to User Properties, you must use the explicit method `Property()`. For example, in the context of Figure 3–1 on page 3-2, `House.FrontWindow1.Property("Position")` refers to the User Property called `Position`. `House.FrontWindow1.Position` instead refers to a child Model node called `Position`.

When referring to System Properties, use the name of the System Property name directly. For example, `FrontWindow1.MinInstances()` refers to the `MinInstances` System Property.

Table 4–12 lists all methods available on Model node identifiers. Return Type indicates the data type of the value returned by the method cited in the Relationship column. Mutable, if Yes, means the value returned is affected by changes in the state of the Model at runtime including instantiation of nodes.

Table 4–12 Property References

Relationship	Applies to	Mutable	Return type	Description
<identifier>.Name()	All model nodes	No	TEXT	Resolves to the model node name of the current identifier.
<identifier>.Description()	All model nodes	No	TEXT	Resolves to the model node description of the current identifier.
<identifier>.Options(),	Option Features, BOM Option Classes, BOM Models	Yes	NODE[]	Resolves to a collection of references to all child model nodes of the current identifier.
<identifier>.Property("<text literal>")	All model nodes	No	BOOLEAN, INTEGER, DECIMAL, or TEXT	Resolves to a reference to the named user-defined property of the current identifier. Return type depends on the type of the user-defined property.

Table 4–12 (Cont.) Property References

Relationship	Applies to	Mutable	Return type	Description
<identifier>.MinInstances() <identifier>.MaxInstances()	Components and BOM models	Yes	INTEGER	Resolves to the dynamic min/max number of instances available at runtime.
<identifier>.InstanceName()	Components and BOM models	No	TEXT	Resolves to the instance name of the current identifier.
<identifier>.Selection()	Features and option classes that have Maximum Number of Selections = 1	Yes	NODE	Resolves to the dynamic child model node of the current identifier that is selected at runtime.
<identifier>.State()	Boolean Features, Option Features, Options, and BOM nodes	Yes	BOOLEAN	Resolves to the dynamic state of the model node.
<identifier>.Value()	Features and BOM nodes	Yes	INTEGER, DECIMAL or TEXT	Resolves to the dynamic value of the model node. Return type depends on the model node type.
<identifier>.Quantity()	Options and BOM nodes	Yes	INTEGER or DECIMAL	Resolves to the dynamic quantity of the model node. Return type depends on the model node type.

The Oracle Configurator parser does not allow the following property references on the left hand side of the rule when using CONTRIBUTE...TO statements:

- <identifier>.Selection.State
- <identifier>.Property("<text literal>")

Example 4–27 shows invalid use of property references used in CONTRIBUTE...TO statements.

Example 4–26 Invalid Property References with CONTRIBUTE...TO Statements

```
CONTRIBUTE a TO b.Selection().State()
CONTRIBUTE a TO b.Property("RequiresGlossyFinish")
```

See the *Oracle Configurator Developer User's Guide* for valid Model structure nodes and System Properties that can be used when defining rules.

4.6.2 Formal Parameters

Formal parameters are local variables defined in rule iterators. They consist of the name of the identifier, prefixed with the ampersand character (&). Each parameter must be unique among the others. Since formal parameters are always prefixed there is no danger of ambiguity with model node references. Model nodes with the same name as a formal parameter (`&win`) must be in quotes when referred to in CDL (`'&win'`).

In Example 4–27, the parameter `&var` is used in the CONTRIBUTE statement. It is declared in the FOR ALL iterator, and it is used in the WHERE clause.

Example 4–27 Formal Parameter

```
CONTRIBUTE &var.Property("NumProp") + 10 TO d
FOR ALL &var IN a.Options()
WHERE &var.Property("prop3") < 5;
```

4.6.2.1 Local Variables and Data Types

Local variables are used exclusively for rule iterators (FOR ALL) and are implicitly declared a data type equivalent to the inferred type of the iterator collection. This allows the Oracle Configurator parser to catch data type errors rather than leaving it to the compiler.

Example 4–28 shows an acceptable use of a local variable of inferred type NODE.

Example 4–28 Valid Local Variable of Inferred Data Type

```
...
CONSTRAIN &Color.selected().property("dark") IMPLIES Frame.Glass.Tint.Dark
FOR ALL &Color in OptionsOf(House);
...
```

Once the inferred type of the local variable is determined, the Oracle Configurator parser can validate its use in the context. For example, a local variable of type NODE can be combined with a Model object identifier to produce a compound reference to a Model node or Property.

4.6.2.2 Local Variables and References

The Oracle Configurator parser allows the reference shown in Example 4–29, but an error displays when you generate logic if `&LocalVar` evaluates to a node or a Property (not the name).

Example 4–29 Valid Formal Parameter and Reference

```
...
&NodeArg.Child(&LocalVar)
...
```

The Oracle Configurator parser does not allow the references shown in Example 4–30. In the first line, a formal parameter can appear only at the beginning of a Model object reference. In the second line, a Property must evaluate to Property value.

Example 4–30 Formal Parameter and an Invalid Reference

```
...
&NodeArg.&LocalVar
&NodeArg.Property(&LocalVar)
...
```

4.6.3 Literals

CDL supports the use of literals of any of the primitive data types:

- Numeric Literals
- Boolean Literals
- Text Literals
- Collection Literals

See Appendix A, "CDL Formal Grammar" for the syntax definition of literals.

4.6.3.1 Numeric Literals

Numeric literals are simply presented as a sequence of digits as in Java.

Table 4–13 Types of Numeric Literals

Numeric Literal	Description
3	Integer literal
128	Integer literal
25.1234	Decimal literal
.01	Decimal literal
6.137E+23	Decimal literal
1e-9	Decimal literal
E	Decimal literal, the constant e
PI	Decimal literal, the constant PI

4.6.3.2 Boolean Literals

Boolean literals are presented by the keywords TRUE and FALSE.

4.6.3.3 Text Literals

Text literals are presented by a sequence of Unicode characters enclosed in double double-quotes ("..."). Comments and whitespace characters are not detected inside text literals. Literal concatenation is allowed using the plus (+) operator - this allows long text literals to be placed on multiple lines. The resulting **terminal** symbol is still returned as a single literal.

Example 4–31 Text Literals

```
...
"This is a text literal"
...
"This text is not a /*comment*/. "+
"All symbols are included in the literal"
...
```

Example 4–32 Text Literal with Escapes

```
...
"This \"text\"\n is quoted and on two lines"
...
```

Example 4–33 Multiple-Line Text Literal

```
...
"This is also a text literal "+
"that continues on this line "+
"and this. It forms one long line of text"
...
```

Table 4–14 shows the escaped characters that can be used inside the double quotes:

Table 4–14 Escaped Characters Inside Double Quotes

Escaped Character	Hexadecimal Value	Abbreviation	Description
`\t`	`\u0009`	HT	horizontal tab
`\n`	`\u000a`	LF	linefeed
`\f`	`\u000c`	FF	form feed
`\r`	`\u000d`	CR	carriage return
`\"`	`\u0022`	"	double quote
`\\`	`\u005c`	\	backslash

4.6.3.4 Collection Literals

Collection literals are not exactly literals in the token sense as they are described in the syntactical grammar. They consist of a sequence of tokens (identifiers or literals) separated by commas (",") and enclosed by braces ("{" and "}"), as shown in Example 4–34 through Example 4–37.

In Example 4–34, Collection 2 shows an element of a collection being specified as another collection. Collections that contain other collections are flattened into a flat list of elements when the rule is compiled. In other words, the content of the inner collection is substituted into the outer collection.

Example 4–34 A Valid Collection of Integer Literals

Collection 1

```
...
{3, 25, 0, -34, 128}
...
```

Collection 2

```
...
{3, 25, {0, -34}, 128}
...
```

Example 4–35 shows several valid collections of Model nodes.

Example 4–35 Valid Collection of Nodes

Collection 1

```
...
{A, B, OptionsOf(C)}
...
```

Collection 2

```
...
{A, B, C1, C2, C3}
...
```

Collection 3

```
...
{MyTotal, MyFeature}
...
```

Collection 4

```
...
{FrontWindow1, FrontWindow3, SideWindow2}
...
```

In Example 4–35, Collection 2 is the same as Collection 1 with exploded children of `C`. And Collections 3 and 4 contain Model node names.

Only collections of homogenous data types are allowed in CDL. That means you cannot mix integer and text literals in a single collection. But you can mix them if the different literals can be implicitly converted to the same type. See Section 2.3, "Data Types" on page 2-4 for more information about implicit conversions. Validation of a homogeneous collection checks that all elements in the collection are valid for all uses of the collection.

As shown in the following examples, the inferred data type of the collection is the least common type of all elements:

In Example 4–36, Collection 1 is a valid collection of decimal literals. Collection 2 is also a valid collection of decimals because `MyTotal` converts to a decimal.

Example 4–36 Valid Collections of Decimals

Collection 1

```
...
{3, 25.0, 1e-9, -34}
...
```

Collection 2

```
...
{MyTotal, {1e-9, -34}}
...
```

Example 4–37 shows an invalid collection because there is no distinct data type that can be inferred.

Example 4–37 Invalid Collection

```
...
{"aha", 25, 128, true}
...
```

4.7 Separators

Separators are characters that serve as syntactic filling between the keywords and the expressions. Their goal is to maintain the structure of the token stream by introducing boundaries between the tokens and by grouping the tokens through some syntactic criteria. See Appendix A, "CDL Formal Grammar" for the syntax definition of separators.

Table 4–15 lists the separators that are valid in CDL.

Table 4–15 Valid CDL Separators

Separator	Description
(	The open parenthesis indicates the beginning of function arguments or the beginning of an expression.
)	The close parenthesis indicates the end of function arguments or the end of an expression
,	The comma separates arguments or collection elements.
;	The semi-colon separates statements.
.	The dot character separates identifiers in compound references.

4.8 Comments and Whitespace

Both comments and the whitespace category of elements are not tokens and therefore ignored by the Oracle Configurator parser.

See Section A, "CDL Formal Grammar" for the syntax definition of comments and whitespace.

4.8.1 Comments

You can add either single-line or multi-line comments to a rule written in CDL. Single-line comments are preceded by two hyphens ("- -") or a two slashes ("//") and end with the new line separator (such as a carriage return or line feed). A multi-line comment is preceded by a slash and an asterisk ("/*"). An asterisk followed by a slash ("*/") indicates the end of the comment.

Example 4–38 shows single-line and multi-line comments.

Example 4–38 CDL Comments

```
-- This is a single-line comment
// This is also a single-line comment
/* This is a multi-line comment,
 spanning across lines */
```

Example 4–39 shows multiple comment lines within a Statement Rule.

Example 4–39 Multiple Line Comments within a Statement Rule

```
/******************
* This constrains the color of the frame
* to the tint of the glass.
*/
BLACK  -- This comes from Frame.Color.Black
IMPLIES
Dark  -- This comes from Glass.Tint.Dark
```

The constraint shown in Example 4–39 can also be written as follows without losing its syntax and semantics:

```
Black IMPLIES Dark
```

See Section A.2.6, "Comment Symbols" on page A-6 for more information.

4.8.2 Whitespace and Line Terminators

Whitespace characters include the following:

- Blank spaces (`' '`)
- Tabs (`'\t'`)
- New lines (`'\n'`)
- Line feed (`'\l'`)
- Carriage return (`'\r'`)
- Form feed (`'\f'`)

A CDL Formal Grammar

This appendix presents the following topics:

- Notation Used in Presenting CDL Grammar
- Terminal Symbols
- Nonterminal Symbols
- EBNF Source Code Definitions for CDL Terminal Symbols

A.1 Notation Used in Presenting CDL Grammar

The notation used in this appendix to present the lexical grammar of CDL follows the Extended Backus-Naur Form (EBNF) symbols. Table A–1 describes the EBNF symbols to help you read this appendix.

Table A–1 Notation Used in Presenting CDL Grammar (EBNF)

Symbol	Description
|	A vertical bar separates alternatives within brackets, braces, or alternative productions.
[]	Square brackets enclose optional items.
{}	Braces enclose repetition.
*	An asterisk shows that the preceding element can be repeated 0 or more times.
+	A plus shows that the preceding element can be repeated 1 or more times.
?	A question mark shows that the preceding element can be repeated 0 or 1 times.
-	A minus shows that the trailing element has been excluded from the preceding element.
:	A colon shows assignment of the production(s) that follow, separated by a vertical bar (|) if multiple.
::=	In Section A.3, "Nonterminal Symbols" on page A-7, a doubled colon followed by an equals sign shows assignment of the production(s) that follow.
#	In Section A.4, "EBNF Source Code Definitions for CDL Terminal Symbols" on page A-9, a pound sign shows that a symbol name is private to the set of terminal symbols.

Table A–1 (Cont.) Notation Used in Presenting CDL Grammar (EBNF)

Symbol	Description
< >	Angle brackets enclose the name of a terminal symbol. In Section A.4 on page A-9, angle brackets also enclose the definition of a terminal symbol.
`TERMINAL`	Terminal symbols represent the names, characters, or literal strings of tokens. Quoted upper case is used for terminal symbols. CONSTRAIN and WHERE are examples of terminal symbols.
`NonTerminal`	Nonterminal symbols represent the names of grammar rules. Unquoted mixed case is used for non-terminals. ConstrainingExpression and BooleanExpression are examples of nonterminal symbols.

The grammar presented in this appendix includes productions containing a nonterminal symbol followed by a sequence of terminal or nonterminal symbols. Alternative sequences start with a vertical bar. For an explanation of syntax typographical conventions and symbols, see also Section 1.4, "Syntax Notation" on page 1-3.

A.1.1 Examples of Notation Used in Presenting CDL Grammar

This section provides examples of the use of the notation described in Table A–1 on page A-1. You can use it to interpret the definitions provided in Section A.2, "Terminal Symbols" on page A-3 and Section A.3, "Nonterminal Symbols" on page A-7.

Example 1

The following definition is from Section A.2.1, "Keyword Symbols" on page A-3:

```
CONSTRAIN
: "CONSTRAIN"
```

This definition means that the the terminal symbol `CONSTRAIN` is defined as the character string `CONSTRAIN`.

Example 2

The following definition is from Section A.2.3, "Literal Symbols" on page A-4:

```
INTEGER_LITERAL
: "0" | <NONZERO_DIGIT> ( <DIGIT> )*
```

This definition means that the the terminal symbol `INTEGER_LITERAL` is defined as:

- the digit `0`, or
- a single occurrence of the symbol **NONZERO_DIGIT** followed by 0 or more occurrences of the symbol **DIGIT**

Example 3

The following definition is from Section A.3, "Nonterminal Symbols" on page A-7:

```
Constraint
::= ( <CONSTRAIN> )? ConstrainingExpression
```

This definition means that the nonterminal symbol `Constraint` is defined as 0 or 1 occurrences of the symbol **CONSTRAIN** followed by the symbol **ConstrainingExpression**.

A.2 Terminal Symbols

This section summarizes the terminal symbols (lexical productions) for CDL, in the form of EBNF. For your convenience in using this section, the names of symbols referenced in another symbols or rule are cross-references linked to their definitions. For example, the cross-reference link **CONSTRAIN** is used in some rules; in that rule you can use the link to jump to the definition of the symbol CONSTRAIN.

The format of the EBNF coding in this section has been edited slightly for easier reading. To examine the precise set of terminal symbol definitions, see Section A.4, "EBNF Source Code Definitions for CDL Terminal Symbols" on page A-9.

See Table A–1, " Notation Used in Presenting CDL Grammar (EBNF)" on page A-1 for informationon the notation used in this section.

A.2.1 Keyword Symbols

See Section 4.3, "Keywords" on page 4-4 for an explanation of this topic.

Example A–1 EBNF for Keyword Symbols

```
CONSTRAIN
: "CONSTRAIN"
CONTRIBUTE
: "CONTRIBUTE"
COMPATIBLE
: "COMPATIBLE"
OF
: "OF"
FORALL
: "FOR ALL"
IN
: "IN"
WHERE
: "WHERE"
COLLECT
: "COLLECT"
DISTINCT
: "DISTINCT"
WHEN
: "WHEN"
WITH
: "WITH"
TO
: "TO"
REQUIRES
: "REQUIRES"
IMPLIES
: "IMPLIES"
EXCLUDES
: "EXCLUDES"
NEGATES
: "NEGATES"
DEFAULTS
: "DEFAULTS"
FUNC_PTR
: "@"
```

A.2.2 Operator Symbols

See Section 4.4, "Operators" on page 4-7 for an explanation of this topic.

Example A–2 EBNF for Operator Symbols

```
PLUS
: "+"
MINUS
: "-"
MULTIPLY
: "*"
DIVIDE
: "/"
ZDIV
: "ZDIV"
MOD
: "%"
EXP
: "^"
EQUALS
: "="
NOT_EQUALS
: "<>"
GT
: ">"
GE
: ">="
LT
: "<"
LE
: "<="
NOT
: "NOT"
NOTTRUE
: "NOTTRUE"
AND
: "AND"
OR
:  "OR"
LIKE
: "LIKE"
```

A.2.3 Literal Symbols

See Section 4.6.3, "Literals" on page 4-22 for an explanation of this topic. Table 4–14, " Escaped Characters Inside Double Quotes" on page 4-23 describes the values of **TEXT_LITERAL**.

Example A–3 EBNF for Literal Symbols

```
END
: "\\0"
DIGITS
: ( <DIGIT> )+
DIGIT
: "0" | <NONZERO_DIGIT>
NONZERO_DIGIT
```

```
: ["1"-"9"]
TEXT_LITERAL
: "\"" ( ~["\"","\\","\n","\r"]
| "\\" ["n","t","b","r","f","\\","\""]
)* "\""
INTEGER_LITERAL
: "0" | <NONZERO_DIGIT> ( <DIGIT> )*
DECIMAL_LITERAL
: ( <INTEGER_LITERAL> "." ( <DIGITS> )? ( <EXPONENTIAL> )?
| <INTEGER_LITERAL> <EXPONENTIAL>
| "." <DIGITS> ( <EXPONENTIAL> )?
| "PI"
| "E"
)
EXPONENTIAL
: "E" ( <PLUS> | <MINUS> )? <INTEGER_LITERAL>
BOOLEAN_LITERAL
: "TRUE"
| "FALSE"
```

A.2.4 Separator Symbols

See Section 4.7, "Separators" on page 4-24 for an explanation of this topic.

Example A–4 EBNF for Separator Symbols

```
DOT
: "."
COMMA
: ","
SEMICOLON
: ";"
LPAREN
: "("
RPAREN
: ")"
LBRACKET
: "{"
RBRACKET
: "}"
```

A.2.5 Identifier Symbols

See the following sections for an explanation of this topic:

- Section 4.6.1, "References" on page 4-17
- Section 4.6.1.1, "Model Object Identifiers" on page 4-17
- Section 4.6.1.4, "Property References" on page 4-19
- Section B.2.1, "Unicode Characters" on page B-3

Table A–2 lists the values allowed in the **LETTER** symbol.

Table A–2 Values for Unicode Escapes Allowed in Identifiers

Unicode	Character
"\u0024"	$ (dollar sign)
"\u0041"-"\u005a"	A through Z
"\u005f"	_ (underscore)
"\u0061"-"\u007a"	a through z
"\u00c0"-"\u00d6"	Latin Capital Letter A With Grave through Latin Capital Letter O With Diaeresis
"\u00d8"-"\u00f6"	Latin Capital Letter O With Stroke through Latin Small Letter O With Diaeresis
"\u00f8"-"\u00ff"	Latin Small Letter O With Stroke through Latin Small Letter Y With Diaeresis
"\u0100"-"\u1fff"	Latin Capital Letter A With Macron through Greek Dasia
"\u3040"-"\u318f"	Hiragana Letter Small A through Hangul Letter Araeae
"\u3300"-"\u337f"	Square Apaato through Square Corporation
"\u3400"-"\u3d2d"	CJK Unified Ideographs
"\u4e00"-"\u9fff"	CJK Unified Ideographs
"\uf900"-"\ufaff"	CJK Compatibility Ideographs

Example A–5 EBNF for Identifier Symbols

```
USER_PROP_IDENTIFIER
: "property"
SIMPLE_IDENTIFIER
: <LETTER> ( <LETTER_OR_DIGIT> )*
FORMAL_IDENTIFIER
: "&" <LETTER> ( <LETTER_OR_DIGIT> )*
QUOTED_IDENTIFIER
: "'" ( ~["\'"] | "\\'")* "'"
LETTER
: ["\u0024", "\u0041"-"\u005a", "\u005f", "\u0061"-"\u007a",
   "\u00c0"-"\u00d6", "\u00d8"-"\u00f6", "\u00f8"-"\u00ff",
   "\u0100"-"\u1fff", "\u3040"-"\u318f", "\u3300"-"\u337f",
   "\u3400"-"\u3d2d", "\u4e00"-"\u9fff", "\uf900"-"\ufaff"]
LETTER_OR_DIGIT
: <LETTER> | <DIGIT> | ( "\\" ( "\"" | "\'" | "\\" ) | "\'" ) >
```

A.2.6 Comment Symbols

See Section 4.8.1, "Comments" on page 4-25 for an explanation of this topic.

Example A–6 EBNF for Comment Symbols

```
"//"
: IN_SINGLE_LINE_COMMENT
"--"
: IN_SINGLE_LINE_COMMENT
```

```
"/*"
: IN_MULTI_LINE_COMMENT
IN_SINGLE_LINE_COMMENT
:
<SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
IN_MULTI_LINE_COMMENT
:
< : "*/" > : DEFAULT
IN_SINGLE_LINE_COMMENT,IN_MULTI_LINE_COMMENT
:
  < ~[] >
```

A.2.7 Whitespace Symbols

See Section 4.8.2, "Whitespace and Line Terminators" on page 4-26 for an explanation of this topic.

Example A–7 EBNF for Whitespace Symbols

```
WHITESPACE
: ( " " | "\t" | "\f" | <LINE_BREAK> )+
LINE_BREAK
: "\n" | "\r" | "\r\n"
```

A.3 Nonterminal Symbols

This section summarizes the nonterminal symbols for CDL, in the form of EBNF. For your convenience in using this section, the names of symbols referenced in another symbols or rule are cross-references linked to their definitions. For example, the cross-reference link **Expression** is used in some rules; in that rule you can use the link to jump to the definition of the symbol `Expression`.

See Table A–1, " Notation Used in Presenting CDL Grammar (EBNF)" on page A-1 for informationon the notation used in this section.

Example A–8 EBNF for Nonterminal Symbols

```
Statements
::= ( ( Statement )? ) ( ";" ( Statement )? )* ( <END> | <EOF> )

Statement
::= ( Constraint | Contribute | Compatible )

Constraint
::= ( <CONSTRAIN> )? ConstrainingExpression

ConstrainingExpression
::= ( Expression ( ( ConstrainingOperator Expression ) ( ForAll )? )? )

ConstrainingOperator
::= ( <REQUIRES> | <IMPLIES> | <EXCLUDES> | <NEGATES> | <DEFAULTS> )

Contribute
::= ( <CONTRIBUTE> Expression <TO> Reference ) ( ForAll )?

Compatible
```

```
::= ( <COMPATIBLE> ( <FORMAL_IDENTIFIER> <OF> Reference ) ( ( "," <FORMAL_
IDENTIFIER> <OF> Reference ) )+ Where )

Method
::= ( <SIMPLE_IDENTIFIER> Arguments )

Event
::= ( <SIMPLE_IDENTIFIER> ( ":" <TEXT_LITERAL> )? )

EventScope
::= ( <SIMPLE_IDENTIFIER> )

ForAll
::= ( <FORALL> Iterator ( "," Iterator )* ( Where )? )

Where
::= ( <WHERE> Expression )

Iterator
::= ( <FORMAL_IDENTIFIER> <IN> ( CollectionExpression | CollectionLiteral |
Function | Reference ) )

Expression
::= OrExpression

OrExpression
::= ( AndExpression ( <OR> OrExpression )? )

AndExpression
::= ( EqualityExpression ( <AND> AndExpression )? )

EqualityExpression
::= ( RelationalExpression ( ( <EQUALS> | <NOT_EQUALS> | <LIKE> )
EqualityExpression )? )

RelationalExpression
::= ( AdditiveExpression ( ( <GT> | <GE> | <LT> | <LE> ) RelationalExpression )? )

AdditiveExpression
::= ( MultiplicativeExpression ( ( ( <PLUS> | <MINUS> ) AdditiveExpression ) )? )

MultiplicativeExpression
::= ( UnaryExpression ( ( ( <MULTIPLY> | <DIVIDE> | <ZDIV> | <MOD> )
MultiplicativeExpression ) )? )

UnaryExpression
::= ( ( ( ( <PLUS> | <MINUS> | <NOT> | <NOTTRUE> ) )? ExponentExpression ) )

ExponentExpression
::= ( PrimaryExpression ( "^" ExponentExpression )? )

PrimaryExpression
::= ( CollectionExpression | Literal | "(" Expression ")" | Function | Reference )

CollectionExpression
::= "{" "COLLECT" ( ( <DISTINCT> ) )? Expression ForAll "}"

Literal
::= ( ( <INTEGER_LITERAL> ) | ( <DECIMAL_LITERAL> ) | ( <BOOLEAN_LITERAL> ) | (
<TEXT_LITERAL> ) | CollectionLiteral )
```

```
Arguments
::= "(" ( ExpressionList )? ")"

ExpressionList
::= ExpressionElement ( "," ExpressionElement )*

ExpressionElement
::= ( ( <FUNC_PTR> ( FunctionName | AnyOperator ) ) | Expression )

CollectionLiteral
::= "{" ( ExpressionList )? "}"

Function
::= ( FunctionName Arguments )
See Section 4.5, "Functions" for a list of available functions.

Reference
::= ( ( ModelIdentifier ( <DOT> ModelIdentifier )* ( <DOT> SysPropIdentifier )* (
<DOT> UserPropIdentifier )? ) | ( ArgumentIdentifier ( <DOT> SysPropIdentifier )*
( <DOT> UserPropIdentifier )? ) )

UserPropIdentifier
::= ( <USER_PROP_IDENTIFIER> "(" <TEXT_LITERAL> ")" )

SysPropIdentifier
::= ( <SIMPLE_IDENTIFIER> Arguments )

ModelIdentifier
::= ( ( <SIMPLE_IDENTIFIER> | <QUOTED_IDENTIFIER> ) )

ArgumentIdentifier
::= <FORMAL_IDENTIFIER>

FunctionName
::= ( <SIMPLE_IDENTIFIER> | <FORMAL_IDENTIFIER> | <QUOTED_IDENTIFIER> )

AnyOperator
::= ( ConstrainingOperator | <CONTRIBUTE> | <COMPATIBLE> | <PLUS> | <MINUS> |
<MULTIPLY> | <DIVIDE> | <ZDIV> | <MOD> | <EXP> | <EQUALS> | <NOT_EQUALS> | <GT> |
<GE> | <LT> | <LE> | <NOT> | <NOTTRUE> | <AND> | <OR> | <LIKE> )
```

A.4 EBNF Source Code Definitions for CDL Terminal Symbols

This section provides the precise set of definitions for the terminal symbols of CDL, taken directly from the defining source code. For a version of these definitions that is edited slightly for easier reading, see Section A.2, "Terminal Symbols" on page A-3.

Example A–9 EBNF Source Code for Terminal Symbols

```
SPECIAL_TOKEN : /* whitespace */
{
  < WHITESPACE: ( " " | "\t" | "\f" | <LINE_BREAK> )+ >
| < LINE_BREAK: "\n" | "\r" | "\r\n" >
}


MORE : /* comments */
```

```
{

  "//" : IN_SINGLE_LINE_COMMENT
| "--" : IN_SINGLE_LINE_COMMENT
| "/*" : IN_MULTI_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN :
{

  <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
}

<IN_MULTI_LINE_COMMENT>
SPECIAL_TOKEN :
{

  <MULTI_LINE_COMMENT: "*/" > : DEFAULT
}

<IN_SINGLE_LINE_COMMENT,IN_MULTI_LINE_COMMENT>
MORE :
{

  < ~[] >
}


TOKEN : /* literals */
{ < END: "\\0" >
| < #DIGITS: ( <DIGIT> )+ >
| < #DIGIT: "0" | <NONZERO_DIGIT> >
| < #NONZERO_DIGIT: ["1"-"9"] >
| < TEXT_LITERAL: "\"" ( ~["\"","\\","\n","\r"]
                       | "\\" ["n","t","b","r","f","\\","\""]
                       )* "\"" >
| < INTEGER_LITERAL: "0" | <NONZERO_DIGIT> ( <DIGIT> )* >
| < DECIMAL_LITERAL: ( <INTEGER_LITERAL> "." ( <DIGITS> )? ( <EXPONENTIAL> )?
                     | <INTEGER_LITERAL> <EXPONENTIAL>
                     | "." <DIGITS> ( <EXPONENTIAL> )?
                     | "PI"
                     | "E"
                     ) >
| < #EXPONENTIAL: "E" ( <PLUS> | <MINUS> )? <INTEGER_LITERAL> >
| < BOOLEAN_LITERAL: "TRUE" | "FALSE" >
}


TOKEN : /* operators */
{
  < PLUS: "+" >
| < MINUS: "-" >
| < MULTIPLY: "*" >
| < DIVIDE: "/" >
| < ZDIV: "ZDIV" >
| < MOD: "%" >
| < EXP: "^" >
| < EQUALS: "=" >
| < NOT_EQUALS: "<>" >
| < GT: ">" >
| < GE: ">=" >
| < LT: "<" >
| < LE: "<=" >
| < NOT: "NOT" >
| < NOTTRUE: "NOTTRUE" >
| < AND: "AND" >
| < OR:  "OR" >
| < LIKE: "LIKE" >
}


TOKEN : /* keywords */
{
  < CONSTRAIN: "CONSTRAIN" >
| < CONTRIBUTE: "CONTRIBUTE" >
| < COMPATIBLE: "COMPATIBLE" >
| < OF: "OF" >
| < FORALL: "FOR ALL" >
| < IN: "IN">
| < WHERE: "WHERE" >
| < COLLECT: "COLLECT" >
| < DISTINCT: "DISTINCT" >
| < WHEN: "WHEN" >
| < WITH: "WITH" >
| < TO: "TO" >
| < REQUIRES: "REQUIRES" >
| < IMPLIES: "IMPLIES" >
| < EXCLUDES: "EXCLUDES" >
| < NEGATES: "NEGATES" >
| < DEFAULTS: "DEFAULTS" >
| < FUNC_PTR: "@" >
}


TOKEN : /* separators */
{
  < DOT: "." >
| < COMMA: "," >
| < SEMICOLON: ";" >
| < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACKET: "{" >
| < RBRACKET: "}" >
}


TOKEN : /* identifiers */
{

  < USER_PROP_IDENTIFIER: "property" >
| < SIMPLE_IDENTIFIER: <LETTER> ( <LETTER_OR_DIGIT> )* >
| < FORMAL_IDENTIFIER: "&" <LETTER> ( <LETTER_OR_DIGIT> )* >
| < QUOTED_IDENTIFIER: "'" ( ~["\'"] | "\\'")* "'" >
| < #LETTER: ["\u0024", "\u0041"-"\u005a", "\u005f", "\u0061"-"\u007a",
             "\u00c0"-"\u00d6", "\u00d8"-"\u00f6", "\u00f8"-"\u00ff",
             "\u0100"-"\u1fff", "\u3040"-"\u318f", "\u3300"-"\u337f",
             "\u3400"-"\u3d2d", "\u4e00"-"\u9fff", "\uf900"-"\ufaff"] >
| < #LETTER_OR_DIGIT: <LETTER> | <DIGIT> | ( "\\" ( "\"" | "\'" | "\\" ) | "\'" )
>
```

```
                    }
```

B CDL Validation

To create correct CDL syntax and understand errors that occur during CDL rule validation, it is helpful to know what the Oracle Configurator parser requires. This appendix provides information on the following topics:

- Validation of CDL
- The Input Stream to the Oracle Configurator Parser
- Name Substitution

B.1 Validation of CDL

As with any language, CDL requires a precise syntax and grammar to ensure that it can be interpreted by Oracle Configurator. The Oracle Configurator parser handles validation at the level of individual rules. A compiler uses the parser to validate the entire set of rules in a configuration model, and then translates them into executable code.

B.1.1 The Parser

The Oracle Configurator parser analyzes the CDL input stream of a rule definition. The parser ignores whitespace and comments and only analyzes the tokens of the input stream to determine if the rule definition is valid.

The parser validates the grammar and structure of rule definition tokens according to the Extended Backus-Naur Form (EBNF). For more information, see Appendix A. The parser is part of the compiler. For details about the compiler, Section B.1.2 on page B-2.

B.1.1.1 Calling the Oracle Configurator Parser

In Configurator Developer, clicking the following buttons in the Statement Rule Details page calls the Oracle Configurator parser:

- Validate Rule Text
- Apply
- Apply and Create Another

B.1.1.2 The Parser's Validation Criteria

Unless the following are true, the parser returns an error:

- All tokens are known CDL elements
- The token order is correct according to CDL syntax

- Data types match within an expression
- Model nodes specified in the rule exist in the Model structure, and can be unambiguously identified by the specified path
- Operators are valid
- Model names that are identical to a CDL keyword are in quotation marks
- Node names containing spaces are in single quotation marks
- Comment statements are properly delimited with a double hyphen, double slash, or /* and */
- A variable (formal identifier) contains an ampersand (&) preceding the variable name
- A variable name is declared only once in the same scope
- A variable appears only once in a single statement (&NodeArg.&LocalVar)

B.1.2 The Compiler

The Oracle Configurator compiler parses all the rule definitions in a Model and then translates the rule set into executable code that can be interpreted by the runtime Oracle Configurator engine.

B.1.2.1 Calling the Oracle Configurator Compiler

The compiler runs when you generate logic for the Model or if UI conditions change during UI Refresh.

For more information about generating logic in Configurator Developer, see the *Oracle Configurator Developer User's Guide*.

For information about generating logic programmatically using CZ_modelOperations_pub.GENERATE_LOGIC and CZ_modelOperations_pub.REFRESH_JRAD_UI, see the *Oracle Configurator Implementation Guide*.

B.1.2.2 The Compiler's Validation Criteria

Unless the following are true, the compiler returns an error:

- Text expressions evaluate to a static string (information that does not change at runtime)
- The data type of arguments match
- The passed parameter resolves to a reference to an existing Model node or Property
- User Properties must be static strings

  For example, `&A.Property("Hi")` but not `&A.Property(B.value)`
- Participants of LIKE and NOT LIKE evaluate to a static string
- All rule participants exist in the Model structure

B.2 The Input Stream to the Oracle Configurator Parser

The input stream presented to the Oracle Configurator parser for lexical analysis is a sequence of Unicode characters. The input stream is processed through the following translations:

- All Unicode escaped characters from the input stream are translated into raw Unicode characters. For information about the format, see Section B.2.1, "Unicode Characters" on page B-3
- All input characters are translated via the lexical rules into lexemes. The lexemes are whitespace characters, comments, and tokens. Whitespace characters and comments are ignored.

  If there are errors in the lexical translation of characters into tokens they will be raised as parser errors.
- Successfully translated tokens are presented for syntactical analysis as a stream of terminal symbols. Tokens in the input stream can be one of the following types
  - Keyword
  - Operator
  - Literal
  - Identifiers
  - Separator.

  Additional details about each token type are provided in Chapter 4, "CDL Elements".

B.2.1 Unicode Characters

Unicode escaped characters are of the format \uxxxx where xxxx is the hexadecimal value representing the character in the Unicode character set. For example the Unicode escape of the character "?" is "\u003f".

B.3 Name Substitution

When parsing identifiers that are references, the Oracle Configurator parser extracts the identity of each identifier (ps_node_id/model_ref_expl_id) and stores it with the intermediate representation of the identifier. This preserves the semantics of the rules regardless of name changes or modifications to the Model structure.

B.3.1 Name Persistency

Since the model object identifiers are case insensitive, the Oracle Configurator parser must preserve the original format of the rule definition. If the Model structure or node names participating in the rule do not change, Configurator Developer displays the original text exactly as it was entered.

If the name of a rule participant changes, Configurator Developer automatically updates the displayed rule definition at the time of viewing or in the Model Report to prevent you from being misdirected to a different or no longer existing Model node.

B.3.2 Ambiguity Resolution

Model structure changes or changes to a node can cause one or more of the references participating in a rule definition to become ambiguous. You must manually resolve ambiguities by inserting or removing identifiers from the reference, as needed.

Glossary

This glossary contains definitions that you may need while working with Oracle Configurator.

API

Application Programming Interface

applet

A Java application running inside a Web browser. *See also* **Java** and **servlet**.

Archive Path

The ordered sequence of **Configurator Extension Archive**s for a **Model** that determines which **Java class**es are loaded for **Configurator Extension**s and in what order.

argument

A data value or object that is passed to a method or a **Java class** so that the method can operate.

ATO

Assemble to Order

ATP

Available to Promise

base node

The **node** in a **Model** that is associated with a **Configurator Extension** Rule. Used to determine the **event** scope for a **Configurator Extension**.

bill of material

A list of Items associated with a parent Item, such as an assembly, and information about how each Item relates to that parent Item.

Bills of Material

The application in Oracle Applications in which you define a **bill of material**.

binding

Part of a **Configurator Extension** Rule that associates a specified event with a chosen **method** of a **Java class**. *See also* **event**.

BOM

See **bill of material**.

BOM item

The **node** imported into **Oracle Configurator Developer** that corresponds to an Oracle **Bills of Material** item. Can be a **BOM Model**, **BOM Option Class node**, or **BOM Standard Item node**.

BOM Model

A model that you import from Oracle **Bills of Material** into **Oracle Configurator Developer**. When you import a BOM Model, effective dates, **ATO** rules, and other data are also imported into Configurator Developer. In Configurator Developer, you can extend the structure of the BOM Model, but you cannot modify the BOM Model itself or any of its attributes.

BOM Model node

The imported **node** in **Oracle Configurator Developer** that corresponds to a **BOM Model** created in Oracle **Bills of Material**.

BOM Option Class node

The imported **node** in **Oracle Configurator Developer** that corresponds to a BOM Option Class created in Oracle **Bills of Material**.

BOM Standard Item node

The imported **node** in **Oracle Configurator Developer** that corresponds to a BOM Standard Item created in Oracle **Bills of Material**.

Boolean Feature

An **element** of a **component** in the **Model** that has two **options**: true or false.

bug

See **defect**.

build

A specific **instance** of an application during its construction. A build must have an install program early in the project so that application **implementers** can **unit test** their latest work in the context of the entire available application.

CDL

See **Constraint Definition Language**.

CIO

See **Oracle Configuration Interface Object (CIO)**.

command event

An **event** that is defined by a character string, which is considered the command for which **listeners** are listening.

Comparison Rule

An **Oracle Configurator Developer** rule type that establishes a relationship to determine the selection state of a logical **Item** (Option, Boolean Feature, or List-of-Options Feature) based on a comparison of two numeric values (numeric **Features**, **Totals**, **Resources**, **Option** counts, or numeric constants). The numeric values being compared can be computed or they can be discrete intervals in a continuous numeric input.

Compatibility Rule

An **Oracle Configurator Developer** rule type that establishes a relationship among **Features** in the Model to control the allowable combinations of **Options**. *See also*, **Property-based Compatibility Rule**.

Compatibility Table

A kind of Explicit Compatibility Rule. For example, a type of compatibility relationship where the allowable combination of **Options** are explicitly enumerated.

component

A piece of something or a configurable element in a **model** such as a **BOM Model**, **Model**, or **Component**.

Component

An element of the **model structure**, typically containing **Features**, that is configurable and instantiable. An **Oracle Configurator Developer** node type that represents a configurable element of a **Model**. Corresponds to one UI screen of selections in a runtime **Oracle Configurator**.

Component Set

An element of the **Model** that contains a number of instantiated **Components** of the same type, where each Component of the set is independently configured.

concurrent program

Executable code (usually written in SQL*Plus or Pro*C) that performs the function(s) of a requested task. Concurrent programs are stored procedures that perform actions such as generating reports and copying data to and from a database.

configuration

A specific set of specifications for a product, resulting from selections made in a runtime **configurator**.

configuration attribute

A characteristic of an **item** that is defined in the **host application** (outside of its inventory of items), in the **Model**, or captured during a **configuration session**. Configuration attributes are inputs from or outputs to the host application at initialization and termination of the configuration session, respectively.

configuration engine

The part of the runtime **Oracle Configurator** that uses **configuration rules** to validate a **configuration**. Compare **generated logic**.

Configuration Interface Object

See **Oracle Configuration Interface Object (CIO)**.

configuration model

Represents all possible configurations of the available **options**, and consists of **model structure** and **rules**. It also commonly includes **User Interface** definitions and **Configurator Extensions**. A configuration model is usually accessed in a **runtime Oracle Configurator window**. *See also* **model**.

configuration rule

A **Logic Rule**, **Compatibility Rule**, **Comparison Rule**, **Numeric Rule**, **Design Chart**, **Statement Rule**, or **Configurator Extension** rule available in **Oracle Configurator Developer** for defining **configurations**. *See also* **rules**.

configuration session

The time from launching or invoking to exiting **Oracle Configurator**, during which **end users** make selections to configure an orderable product. A configuration session is limited to one **configuration model** that is loaded when the session is initialized.

configurator

The part of an application that provides custom configuration capabilities. Commonly, a window that can be launched from a host application so **end users** can make selections resulting in valid **configurations**. *Compare* **Oracle Configurator**.

Configurator Extension

An extension to the **configuration model** beyond what can be implemented in Configurator Developer.

A type of **configuration rule** that associates a **node**, **Java class**, and event **binding** so that the rule operates when an **event** occurs during a **configuration session**.

A **Java class** that provides methods that can be used to perform configuration actions.

Configurator Extension Archive

An **object** in the **Repository** that stores one or more compiled **Java class**es that implement **Configurator Extension**s.

connectivity

The connection between client and database that allows data communication.

The connection across components of a model that allows modeling such products as networks and material processing systems.

Connector

The **node** in the **model structure** that enables an **end user** at **runtime** to connect the Connector node's parent to a referenced **Model**.

Constraint Definition Language

A language for entering **configuration rule**s as text rather than assembling them interactively in Oracle Configurator Developer. CDL can express more complex constraining relationships than interactively defined configuration rules can.

Container Model

A type of **BOM Model** that you import from Oracle **Bills of Material** into **Oracle Configurator Developer** to create configuration models containing **connectivity** and trackable components. Configurations created from Container Models can be tracked and updated in Oracle Install Base

Contributes to

A relation used to create a specific type of **Numeric Rule** that accumulates a total value. *See also* **Total**.

Consumes from

A relation used to create a specific type of **Numeric Rule** that decrementing a total value, such as specifying the quantity of a **Resource** used.

count

The number or quantity of something, such as selected **options**. *Compare* **instance**.

CTO

Configure to Order

customer

The person for whom products are configured by **end users** of the **Oracle Configurator** or other **ERP** and CRM applications. Also the end users themselves directly accessing **Oracle Configurator** in a Web store or kiosk.

customer requirements

The needs of the customer that serve as the basis for determining the configuration of products, **systems**, and services. Also called needs assessment. *See* **guided buying or selling**.

CZ

The product shortname for **Oracle Configurator** in Oracle Applications.

CZ schema

The implementation version of the standard runtime **Oracle Configurator** data-warehousing schema that manages data for the **configuration model**. The implementation schema includes all the data required for the **runtime** system, as well as specific tables used during the construction of the **configurator**.

data import

Populating the **CZ schema** with enterprise data from **ERP** or legacy systems via **import tables**.

data source

A programmatic reference to a database. Referred to by a data source name (DSN).

DBMS

Database Management System

default

A predefined value. In a **configuration**, the automatic selection of an **option** based on the **preselection** rules or the selection of another option.

Defaults relation

An **Oracle Configurator Developer** Logic Rule relation that determines the logic state of **Features** or **Options** in a default relation to other Features and Options. For example, if A Defaults B, and you select A, B becomes Logic True (selected) if it is available (not Logic False).

defect

A failure in a product to satisfy the **users'** requirements. Defects are prioritized as critical, major, or minor, and fixes range from corrections or workarounds to enhancements. Also known as a bug.

Design Chart

An **Oracle Configurator Developer** rule type for defining advanced Explicit Compatibilities interactively in a table view.

developer

The person who uses **Oracle Configurator Developer** to create a **configurator**. *See also* **implementer** and **user**.

Developer

The tool (**Oracle Configurator Developer**) used to create **configuration models**.

DHTML

Dynamic Hypertext Markup Language

discontinued item

A discontinued item is one that exists in an installed configuration of a component (as recorded in Oracle Install Base), but has been removed from the instance of the component being reconfigured, either by deletion or by deselection.

element

Any entity within a **model**, such as **Options**, **Totals**, **Resources**, UI controls, and **components**.

end user

The ultimate user of the runtime **Oracle Configurator**. The types of end users vary by project but may include salespeople or distributors, administrative office staff, marketing personnel, order entry personnel, product engineers, or customers directly accessing the application via a Web browser or kiosk. *Compare* **user**.

enterprise

The **systems** and **resources** of a business.

environment

The arena in which software tools are used, such as operating system, applications, and **server** processes.

ERP

Enterprise Resource Planning. A software system and process that provides automation for the customer's back-room operations, including order processing.

event

An action or condition that occurs in a **configuration session** and can be detected by a **listener**. Example events are a change in the value of a **node**, the creation of a component **instance**, or the saving of a **configuration**. The part of **model structure** inside which a **listener** listens for an event is called the event **binding** scope. The part of model structure that is the source of an event is called the event execution scope. *See also* **command event**.

Excludes relation

An **Oracle Configurator Developer Logic Rule** type that determines the logic state of **Features** or **Options** in an excluding relation to other Features and Options. For example, if A Excludes B, and if you select A, B becomes Logic False, since it is not allowed when A is true (either User or Logic True). If you deselect A (set to User

False), there is no effect on B, meaning it could be User or Logic True, User or Logic False, or **Unknown**. *See* **Negates relation**.

feature

A characteristic of something, or a configurable element of a **component** at **runtime**.

Feature

An element of the **model structure**. Features can either have a value (numeric or Boolean) or enumerated **Options**.

functional specification

Document describing the functionality of the application based on **user** requirements.

generated logic

The compiled structure and rules of a **configuration model** that is loaded into memory on the Web server at **configuration session** initialization and used by the **Oracle Configurator engine** to validate runtime selections. The logic must be generated either in **Oracle Configurator Developer** or programmatically in order to access the configuration model at **runtime**.

guided buying or selling

Needs assessment questions in the **runtime** UI to guide and facilitate the configuration process. Also, the **model structure** that defines these questions. Typically, guided selling questions trigger **configuration rule** that automatically select some product **options** and exclude others based on the **end user's** responses.

host application

An application within which **Oracle Configurator** is embedded as integrated functionality, such as Order Management or *i*Store.

HTML

Hypertext Markup Language

implementation

The stage in a project between defining the problem by selecting a configuration technology vendor, such as Oracle, and deploying the completed configuration application. The implementation stage includes gathering requirements, defining test cases, designing the application, constructing and testing the application, and delivering it to **end users**. *See also* **developer** and **user**.

implementer

The person who uses **Oracle Configurator Developer** to build the **model structure**, **rules**, and UI customizations that make up a **runtime** Oracle Configurator. Commonly also responsible for enabling the integration of **Oracle Configurator** in a **host application**.

Implies relation

An **Oracle Configurator Developer Logic Rule** type that determines the logic state of **Features** or **Options** in an implied relation to other Features and Options. For example, if A Implies B, and you select A, B becomes Logic True. If you deselect A (set to User False), there is no effect on B, meaning it could be User or Logic True, User or Logic False, or **Unknown**. *See* **Requires relation**.

import server

A database **instance** that serves as a source of data for **Oracle Configurator**'s Populate, Refresh, and Synchronization concurrent processes. The import server is sometimes referred to as the remote server.

import tables

Tables mirroring the CZ schemaItem Master structure, but without integrity constraints. Import tables allow batch population of the CZ schema's Item Master. Import tables also store extractions from Oracle Applications or **legacy data** that create, update, or delete records in the CZ schema **Item Master**.

initialization message

The **XML** message sent from a **host application** to the **Oracle Configurator Servlet**, containing data needed to initialize the runtime Oracle Configurator. *See also* **termination message**.

Instance

An **Oracle Configurator Developer** attribute of a **component's** **node** that specifies a minimum and maximum value. *See also* **instance**.

instance

A **runtime** occurrence of a **component** in a configuration. *See also* **instantiate**. *Compare* **count**.

Also, the memory and processes of a database.

instantiate

To create an instance of something. Commonly, to create an **instance** of a **component** in the runtime **user interface** of a **configuration model**.

integration

The process of combining multiple software **components** and making them work together.

integration testing

Testing the interaction among software programs that have been integrated into an application or **system**. Also called system testing. *Compare* **unit test**.

item

A product or part of a product that is in inventory and can be delivered to customers.

Item

A Model or part of a Model that is defined in the **Item Master**. Also data defined in Oracle Inventory.

Item Master

Data stored to structure the Model. Data in the **CZ schema** Item Master is either entered manually in **Oracle Configurator Developer** or imported from Oracle Applications or a legacy system.

Item Type

Data used to classify the Items in the Item Master. Item Catalogs imported from Oracle Inventory are Item Types in **Oracle Configurator Developer**.

Java

An object-oriented programming language commonly used in internet applications, where Java applications run inside Web browsers and **servers**. Used to implement the behavior of **Configurator Extension**s. *See also* **applet** and **servlet**.

Java class

The compiled version of a **Java** source code file. The **method**s of a Java class are used to implement the behavior of **Configurator Extension**s.

JavaServer Pages

Web pages that combine static presentation elements with dynamic content that is rendered by Java **servlet**s.

JSP

See **JavaServer Pages**.

legacy data

Data that cannot be imported without creating custom extraction programs.

listener

A class in the **CIO** that detects the occurrence of specified **event**s in a **configuration session**.

load

Storing the **configuration model** data in the **Oracle Configurator Servlet** on the Web server. Also, the time it takes to initialize and display a configuration model if it is not preloaded.

The burden of transactions on a **system**, commonly caused by the ratio of **user** connections to CPUs or available memory.

log file

A file containing errors, warnings, and other information that is output by the running application.

Logic Rule

An **Oracle Configurator Developer** rule type that expresses constraint among model elements in terms of logic relationships. Logic Rules directly or indirectly set the logical state (User or Logic True, User or Logic False, or **Unknown**) of **Features** and **Options** in the Model.

There are four primary Logic Rule relations: Implies, Requires, Excludes, and Negates. Each of these rules takes a list of Features or Options as operands. *See also* **Implies relation**, **Requires relation**, **Excludes relation**, and **Negates relation**.

maintainability

The characteristic of a product or process to allow straightforward **maintenance**, alteration, and extension. Maintainability must be built into the product or process from inception.

maintenance

The effort of keeping a **system** running once it has been deployed, through **defect** fixes, procedure changes, infrastructure adjustments, data replication schedules, and so on.

Metalink

Oracle's technical support Web site at:

`http://www.oracle.com/support/metalink/`

method

A function that is defined in a **Java class**. Methods perform some action and often accept parameters.

Model

The entire hierarchical "tree" view of all the data required for **configurations**, including **model structure**, variables such as **Resources** and **Totals**, and elements in support of intermediary rules. Includes both imported **BOM Models** and Models created in Configurator Developer. May consist of BOM Option Classes and BOM Standard Items.

model

A generic term for data representing products. A model contains **elements** that correspond to **items**. Elements may be **components** of other objects used to define products. A **configuration model** is a specific kind of model whose elements can be configured by accessing an **Oracle Configurator window**.

model-driven UI

The graphical views of the **model structure** and **rules** generated by **Oracle Configurator Developer** to present **end users** with interactive product selection based on **configuration models**.

model structure

Hierarchical "tree" view of data composed of **elements** (**Models**, **Components**, **Features**, **Options**, **BOM Models**, **BOM Option Class nodes**, **BOM Standard Item nodes**, **Resources**, and **Totals**). May include reusable **components** (**References**).

Negates relation

A type of **Oracle Configurator Developer Logic Rule** type that determines the logic state of **Features** or **Options** in a negating relation to other Features and Options. For example, if one **option** in the relationship is selected, the other option must be Logic False (not selected). Similarly, if you deselect one option in the relationship, the other option must be Logic True (selected). *See* **Excludes relation**.

node

The icon or location in a **Model** tree in **Oracle Configurator Developer** that represents a **Component**, **Feature**, **Option** or variable (**Total** or **Resource**), **Connector**, **Reference**, **BOM Model**, **BOM Option Class node**, or **BOM Standard Item node**.

Numeric Rule

An **Oracle Configurator Developer** rule type that expresses constraint among model elements in terms of numeric relationships. *See also*, **Contributes to** and **Consumes from**.

object

Entities in **Oracle Configurator Developer**, such as **Models**, Usages, Properties, Effectivity Sets, UI Templates, and so on. *See also* **element**.

OC

See **Oracle Configurator**.

OCD

See **Oracle Configurator Developer**.

option

A logical selection made in the Model Debugger or a runtime Oracle Configurator by the **end user** or a rule when configuring a **component**.

Option

An element of the **Model**. A choice for the value of an enumerated **Feature**.

Oracle Configuration Interface Object (CIO)

A **server** in the **runtime** application that creates and manages the interface between the client (usually a **user interface**) and the underlying representation of **model structure** and **rules** in the **generated logic**.

The CIO is the **API** that supports creating and navigating the Model, querying and modifying selection states, and saving and restoring **configurations**.

Oracle Configurator

The product consisting of development tools and **runtime** applications such as the **CZ schema**, **Oracle Configurator Developer**, and runtime Oracle Configurator. Also the runtime Oracle Configurator variously packaged for use in networked or Web deployments.

Oracle Configurator architecture

The three-tier **runtime** architecture consists of the **User Interface**, the **generated logic**, and the **CZ schema**. The application development architecture consists of **Oracle Configurator Developer** and the CZ schema, with test instances of a runtime **Oracle Configurator**.

Oracle Configurator Developer

The suite of tools in the **Oracle Configurator** product for constructing and maintaining **configurators**.

Oracle Configurator engine

The part of the **Oracle Configurator** product that validates runtime selections. *See also* **generated logic**.

Oracle Configurator schema

See **CZ schema**.

Oracle Configurator Servlet

A **Java** servlet that participates in rendering Legacy user interfaces for **Oracle Configurator**.

Oracle Configurator window

The **user interface** that is launched by accessing a **configuration model** and used by **end users** to make the selections of a **configuration**.

performance

The operation of a product, measured in throughput and other data.

Populator

An entity in **Oracle Configurator Developer** that creates **Component**, **Feature**, and **Option nodes** from information in the **Item Master**.

preselection

The default state in a **configurator** that defines an initial selection of **Components**, **Features**, and **Options** for configuration.

A process that is implemented to select the initial element(s) of the **configuration**.

product

Whatever is ordered and delivered to customers, such as the output of having configured something based on a model. Products include intangible entities such as services or contracts.

Property

A named value associated with a **node** in the **Model** or the **Item Master**. A set of Properties may be associated with an Item Type. After importing a BOM Model, Oracle Inventory Catalog Descriptive Elements are Properties in **Oracle Configurator Developer**.

Property-based Compatibility Rule

An **Oracle Configurator Developer** Compatibility Rule type that expresses a kind of compatibility relationship where the allowable combinations of **Options** are specified implicitly by relationships among Property values of the Options.

prototype

A construction technique in which a preliminary version of the application, or part of the application, is built to facilitate **user** feedback, prove feasibility, or examine other implementation issues.

PTO

Pick to Order

publication

A unique deployment of a **configuration model** (and optionally a **user interface**) that enables a developer to control its availability from host applications such as Oracle Order Management or *i*Store. Multiple publications can exist for the same configuration model, but each publication corresponds to only one **Model** and **User Interface**.

publishing

The process of creating a **publication** record in **Oracle Configurator Developer**, which includes specifying applicability parameters to control **runtime** availability and running an Oracle Applications concurrent process to copy data to a specific database.

RDBMS

Relational Database Management System

reference

The ability to reuse an existing **Model** or **Component** within the structure of another Model (for example, as a subassembly).

Reference

An **Oracle Configurator Developer** node type that denotes a **reference** to another **Model**.

Repository

Set of pages in **Oracle Configurator Developer** that contains areas for organizing and maintaining **Model**s and shared **object**s in a single location.

Requires relation

An **Oracle Configurator Developer** Logic Rule relationship that determines the logic state of **Features** or **Options** in a requirement relation to other Features and Options. For example, if A Requires B, and if you select A, B is set to Logic True (selected). Similarly, if you deselect A, B is set to Logic False (deselected). See **Implies relation**.

resource

Staff or equipment available or needed within an enterprise.

Resource

A variable in the **Model** used to keep track of a quantity or supply, such as the amount of memory in a computer. The value of a Resource can be positive or zero, and can have an Initial Value setting. An error message appears at **runtime** when the value of a Resource becomes negative, which indicates it has been over-consumed. Use **Numeric Rules** to contribute to and consume from a Resource.

Also a specific node type in **Oracle Configurator Developer**. *See also* **node**.

reusable component

See **reference** and **model structure**.

reusability

The extent to and ease with which parts of a **system** can be put to use in other systems.

rules

Also called business rules or **configuration rule**. In the context of Oracle Configurator and **CDL**, a rule is not a "business rule." Constraints applied among elements of the product to ensure that defined relationships are preserved during configuration. Elements of the product are **Components**, **Features**, and **Options**. Rules express logic, numeric parameters, implicit compatibility, or explicit compatibility. Rules provide **preselection** and **validation** capability in **Oracle Configurator**.

See also **Comparison Rule**, **Compatibility Rule**, **Design Chart**, **Logic Rule** and **Numeric Rule**.

runtime

The environment and context in which applications are run, tested, or used, rather than developed.

The environment in which an **implementer** (tester), **end user**, or **customer** configures a product whose model was developed in **Oracle Configurator Developer**. *See also* **configuration session**.

schema

The tables and objects of a data model that serve a particular product or business process. *See also* **CZ schema**.

server

Centrally located software processes or hardware, shared by client**s**.

servlet

A Java application running inside a Web server. *See also* **Java**, **applet**, and **Oracle Configurator Servlet**.

solution

The deployed **system** as a response to a problem or problems.

SQL

Structured Query Language

Statement Rule

An **Oracle Configurator Developer** rule type defined by using the Oracle Configurator **Constraint Definition Language** (text) rather than interactively assembling the rule's elements.

system

The hardware and software **component**s and infrastructure integrated to satisfy functional and **performance** requirements.

termination message

The **XML** message sent from the **Oracle Configurator Servlet** to a **host application** after a **configuration session**, containing configuration outputs. *See also* **initialization message**.

Total

A variable in the **Model** used to accumulate a numeric total, such as total price or total weight.

Also a specific node type in **Oracle Configurator Developer**. *See also* **node**.

UI

See **User Interface**.

UI Templates

Templates available in **Oracle Configurator Developer** for specifying UI definitions.

Unknown

The logic state that is neither true nor false, but unknown at the time a **configuration session** begins or when a Logic Rule is executed. This logic state is also referred to as Available, especially when considered from the point of view of the **runtime Oracle Configurator end user**.

unit test

Execution of individual routines and modules by the application **implementer** or by an independent test consultant to find and resolve **defect****s** in the application. *Compare* **integration testing**.

update

Moving to a new version of something, independent of software release. For instance, moving a production **configurator** to a new version of a **configuration model**, or changing a **configuration** independent of a model **update**.

upgrade

Moving to a new release of **Oracle Configurator** or **Oracle Configurator Developer**.

user

The person using a product or system. Used to describe the person using **Oracle Configurator Developer** tools and methods to build a **runtime Oracle Configurator**. *Compare* **end user**.

User Interface

The part of an **Oracle Configurator** implementation that provides the graphical views necessary to create **configurations** interactively. A **user interface** is generated from the **model structure**. It interacts with the model definition and the **generated logic** to give **end users** access to customer requirements gathering, product selection, and any extensions that may have been implemented. *See also* **UI Templates**.

user interface

The visible part of the application, including menus, dialog boxes, and other on-screen elements. The part of a **system** where the **user** interacts with the software. Not necessarily generated in **Oracle Configurator Developer**. *See also* **User Interface**.

user requirements

A description of what the **configurator** is expected to do from the **end user's** perspective.

validation

Tests that ensure that configured **components** will meet specific criteria set by an enterprise, such as that the components can be ordered or manufactured.

variable

Parts of the **Model** that are represented by **Totals**, **Resources**, or numeric **Features**.

verification

Tests that check whether the result agrees with the specification.

Web

The portion of the Internet that is the World Wide Web.

Workbench

Set of pages in **Oracle Configurator Developer** for creating, editing, and working with **Repository objects** such as **Models** and **UI Templates**.

XML

Extensible Markup Language, a highly flexible markup language for transferring data between **Web** applications. Used for the **initialization message** and **termination message** of the **Oracle Configurator Servlet**.

Index

Symbols

+ *See* addition
{} *See* braces
[] *See* brackets
: *See* colon
, *See* comma
*/ *See* comments
/* *See* comments
// *See* comments
 - *See* comments
+ *See* concatenation
/ *See* division
. *See* dot
' ' *See* double quotes
= *See* equals
- *See* exclusion
> *See* greater than
>= *See* greater than or equal
< *See* less than
<= *See* less than or equal
* *See* multiplication
 ::= *See* NonTerminal symbols
!= *See* not equal
<> *See* not equal
() *See* parentheses
% *See* percent
^ *See* power
* *See* repetition
+ *See* repetition
? *See* repetition
; *See* semi-colon
- *See* subtraction
 <> *See* Terminal symbols
# *See* Terminal symbols
- *See* unary minus
+ *See* unary plus
| *See* vertical bar

A

Abs
  arithmetic function, 4-12
ACos
  trigonometric function, 4-14
adding
  literals concatenation, 4-22
  text concatenation, 4-9
addition
  concatenation, 4-22
addition (operator)
  description, 4-9
  precedence, 4-10
AllTrue logical function
  definition, 4-14
  example, 3-4
AND (logical operator)
  description, 4-8
  precedence, 4-10
AnyTrue logical function
  defined, 4-14
Arithmetic
  operator type, 4-9
arithmetic
  CDL functions, 4-12
ASin
  trigonometric function, 4-14
ATan
  trigonometric function, 4-14
ATan2
  trigonometric function, 4-14

B

BeginsWith
  text function, 4-15
BOOLEAN data type, 2-4
Boolean literals
  See literals
braces (CDL separator)
  collections, 4-23
    example, 4-6
  usage, A-1
brackets (CDL separator)
  usage, A-1

C

Cartesian product
  definition, 1-2
  iterator statements, 4-2
  restriction in CDL, 4-11

case-sensitivity
  constants, 2-4
  formal parameters, 1-3, 2-4
  identifiers, 1-3, 2-4, B-3
  iterators, 1-3
  keywords, 2-4
  literals, 2-4
    Boolean, 2-4
    text, 2-4
  Model nodes, 2-4, B-3
  operands, 2-4
  predefined CDL keywords, 1-3
CDL (Constraint Definition Language)
  conditional expressions, 4-6, 4-7
  considerations, 2-1
  expressing
    configuration rules, 1-1
  flexibility, 3-3
  overview, 1-1
  precedence of operators, 4-9
  principles, 2-1
  readability, 2-3
  rule anatomy, 2-2
  terminology, 1-2
  unavailable rule relationships, 1-2
CDL functions
  arithmetic, 4-12
  logical, 4-14
  set, 4-15
  text, 4-15
  trigonometric, 4-13
CDL keywords
  case sensitivity, 2-4
  COLLECT, 4-10
    with DISTINCT, 4-11
  COMPATIBLE, 4-5
    explicit statements, 4-1
  CONSTRAIN
    explicit statements, 4-1
    keyword operators, 4-4
    Logic and Comparison Rules, 2-2
    restriction, 2-5
    text functions, 4-15
  CONTRIBUTE, 4-5
    explicit statements, 4-1
  DEFAULTS, 4-4, 4-8, 4-10
  DISTINCT, 4-11
  EXCLUDES, 4-4, 4-8, 4-10
  FOR ALL...IN, 4-2, 4-6
  IMPLIES, 4-4, 4-8, 4-10
  NEGATES, 4-4, 4-8, 4-10
  predefined, 1-3
  REQUIRES, 4-4, 4-8, 4-10
  WHERE, 3-4, 4-7
CDL separators
  -, A-1
  *, A-1
  +, A-1
  :, A-1
  ::=, A-1
  ?, A-1
  [], A-1
  {}, A-1
  |, A-1
Ceiling
  arithmetic function, 4-13
characters
  whitespace, 4-22, 4-26
clause
  definition, 1-2
COLLECT (CDL keyword)
  static operands, 4-7
COLLECT (operator)
  iterator statement, 4-10
  with Property values, 4-11
collection literals
  See literals
collections
  CDL operands, 1-2
  COLLECT (operator), 4-11
  definition, 1-2
  valid data types, 4-24
colon
  assignment, A-1
comma (CDL separator)
  collections, 4-23
  definition, 4-25
  function arguments, 4-9, 4-12
comments
  adding to rule definition, 4-25
  definition, 2-3
  detection in text literals, 4-22
  multi-line, 4-25
  rule definitions, 4-25
  single-line, 4-25
  validation, B-2
Comparison Rules
  CDL operators, 4-8
COMPATIBLE (CDL keyword)
  explicit statements, 4-1
  Property-based Compatibility Rule
    representation, 4-5
compiler
  See Oracle Configurator compiler
concatenation (operator)
  description, 4-9
  static usage, 4-10
constants
  case sensitivity, 2-4
CONSTRAIN (CDL keyword)
  constraint statement, 4-4
  explicit statement, 4-1
  restriction, 2-5, 4-15
  use, 2-2
Constraint Definition Language
  See CDL (Constraint Definition Language)
CONTRIBUTE (CDL keyword)
  explicit statements, 4-1
  expression example, 4-3
  Numeric Rule representation, 4-5

Cos
  trigonometric function, 4-14
Cosh
  trigonometric function, 4-14
Count set function
  definition, 4-15

D

data types
  rules, 2-4
  validation, B-2
debugging
  log files, xiv
DECIMAL data type, 2-4
DEFAULTS (logical keyword operator)
  constraint statement, 4-4
  definition, 4-8
  precedence, 4-10
design questions
  CDL rules, 2-1
  Statement Rules, 2-1
designing
  rules, 2-1, 3-4
division (operator)
  description, 4-9
  precedence, 4-9
dot (CDL separator)
  precedence, 4-9
  use in identifiers, 4-9
double quotes
  around literals, 4-22

E

E (CDL numeric constant), 2-4, 4-22
EndsWith (operator)
  text function, 4-15
equals (operator)
  description, 4-8
  precedence, 4-10
  text function, 4-15
errors
  troubleshooting, xiv
EXCLUDES (logical keyword operator)
  constraint statements, 4-4
  definition, 4-8
  precedence, 4-10
exclusion
  minus, A-1
Exp
  arithmetic function, 4-13
explicit statements
  compared to iterator, 2-1
  definition, 1-2
  how to use, 4-1
expressions
  conditional, 4-6, 4-7
  definition, 1-2, 4-3
  equivalency, 3-3
  precedence based on operator, 4-7

F

FALSE (Boolean literal keyword), 2-4, 4-22
Floor
  arithmetic function, 4-13
FOR ALL...IN (CDL keyword)
  iterator statement, 4-2, 4-6
formal parameters
  case sensitivity, 2-4
functions
  arithmetic, 4-12
  CDL, 4-12
  logical, 4-14
  set, 4-15
  text, 4-15
  trigonometric, 4-13

G

greater than (operator)
  description, 4-8
  precedence, 4-10
greater than or equal (operator)
  description, 4-8
  precedence, 4-10

H

heirarchy
  CDL function OptionsOf, 4-15

I

identifiers
  case sensitivity, 1-3, 2-4, B-3
  definition, 1-2
  Model object, 4-17
IMPLIES (logical keyword operator)
  constraint statements, 4-4
  definition, 4-8
  precedence, 4-10
importing
  rules, 1-1
INTEGER data type, 2-4
iterator statements
  advantage of using, 3-3
  Cartesian product, 4-2
  compared to explicit, 2-1
  definition, 1-2, 4-2
  local variables, 4-21
  multiple, 4-2
iterators
  case-sensitivity, 1-3
  Property-based Compatibility Rules, 4-2

K

keywords
  See CDL keywords

L

less than (operator)
  description, 4-8
  precedence, 4-10
less than or equal (operator)
  description, 4-8
  precedence, 4-10
LIKE (operator)
  description, 4-8
  precedence, 4-10
  usage, 4-10
literals
  Boolean operand, 4-22
  case sensitivity, 2-4
  collection, 4-11
  collection operand, 4-23
  concatenation, 4-22
  numeric operand, 4-22
  text
    case sensitivity, 2-4
    concatenation, 4-10
    containing comments, 4-22
    with escapes, 4-22
    multiple lines, 4-22
  text operand, 4-22
  types, 4-22
Log
  arithmetic function, 4-13
log files
  troubleshooting errors, xiv
Log10
  arithmetic function, 4-13
logic
  generating
    OptionsOf function, 3-4
logical
  functions
    CDL, 4-14
  operator type, 4-8

M

maintenance
  rule design, 2-1
Matches
  text function, 4-15
Max
  arithmetic function, 4-13
Max set function
  definition, 4-15
messages
  function overflows and underflows, 4-16
Min
  arithmetic function, 4-13
Min set function
  defined, 4-15
Mod
  arithmetic function, 4-13
Model structure
  identifiers, 4-17
  nodes
    use in rules, 2-1
Models
  design
    structure changes, 2-1, 4-2
  identification in rules, 2-2
multiplication (operator)
  description, 4-9
  precedence, 4-9

N

NEGATES (logical keyword operator)
  constraint statements, 4-4
  definition, 4-8
  precedence, 4-10
NoMatches
  text function, 4-15
NonTerminal symbols
  ::=, A-1
  Constraint, A-2
  definition, A-2
NOT (operator)
  description, 4-8
  precedence, 4-9
not equal (operator)
  description, 4-8
  precedence, 4-10
NOT LIKE (operator)
  description, 4-8
  precedence, 4-10
  usage, 4-10
NotEquals
  text function, 4-15
NotTrue logical function
  CDL operator, 4-8
  definition, 4-14
  precedence, 4-9

O

operands
  case sensitivity, 2-4
  definition, 4-17
  References, 4-17
operators
  Arithmetic, 4-9
  comparison, 4-8
  definition, 4-7
  Logical, 4-8
  precedence, 4-9
  validation, B-2
OptionsOf compound function
  definition, 4-15
  iterator statement, 4-2
  useage of logic generation, 3-4
OR (logical operator)
  description, 4-8
  precedence, 4-10
Oracle Configurator

log files, xiv
TAR template, xiii
Oracle Configurator compiler
definition, 1-2
validation criteria, B-2
Oracle Configurator Developer
importing data to, 1-1
log files, xiv
product support, xiii
Oracle Configurator parser
definition, 1-2
Statement Rules, B-1
validating, B-1
validation criteria, B-1

P

parameters
formal definition, 4-20
parentheses (CDL separator)
function arguments, 4-12
precedence, 4-9
use in expressions, 4-9, 4-25
parser
See Oracle Configurator parser
percent (operator)
description, 4-9
precedence, 4-9
PI (CDL numeric constant), 2-4, 4-22
Pow
arithmetic function, 4-13
power (operator)
description, 4-9
precedence, 4-9
precedence of operators, 4-9
Product Support, xiii
product support for Oracle Configurator Developer, xiii
Properties
example, 3-3
Property-based Compatibility Rules, 4-6
referring to, 4-19
rules, 2-1
System Properties, 4-19
User Properties, 2-4, 4-7, B-2
Property-based Compatibility Rules
CDL, 4-6
evaluation, 4-7
iterators, 4-2

R

References
compound Model nodes, 4-18
compound Properties, 4-19
Model structure, 4-17
operands, 4-17
path, 4-18
relationships
CDL keywords, 2-2
definition, 1-2, 2-1
repetition
asterisk, A-1
braces, A-1
plus, A-1
question mark, A-1
REQUIRES (logical keyword operator)
constraint statements, 4-4
definition, 4-8
precedence, 4-10
Round
arithmetic function, 4-12
RoundDownToNearest
arithmetic function, 4-12
RoundToNearest
arithmetic function, 4-13
RoundUpToNearest
arithmetic function, 4-13
rules
anatomy, 3-3
in CDL, 1-1
CDL rule definition, 2-2
data types in rule definitions, 2-4
description, 2-2
designing, 2-1, 3-4
format of input, 2-3
Model association, 2-2
naming, 2-2
persistent constraints, 4-2
subexpressions
grouping, 4-9, 4-25
rolled up, 3-3
validation, B-2

S

scientific E (CDL numeric constant), 2-4, 4-22
semi-colon (CDL separator)
separating statements, 2-3, 4-25
separators
definition, 4-24
signatures
definition, 1-2
Sin
trigonometric function, 4-14
singleton
definition, 1-2
SinH
trigonometric function, 4-14
Sqrt
arithmetic function, 4-13
Statement Rules
defining, 2-2
definition, 1-1
Oracle Configurator parser, B-1
statements
constraint, 4-4
contributes, 4-5
definition, 1-2, 4-1
explicit, 1-2, 4-1

explicit versus iterator, 2-1, 4-1
iterators
Cartesian product, 4-2
compared to explicit, 2-1
definition, 1-2, 4-2
multiple, 4-2
multiple in one CDL rule definition, 2-3
subtraction (operator)
description, 4-9
precedence, 4-10
Support, xiii
System Properties
referring to, 4-19

T

Tan
trigonometric function, 4-14
TanH
trigonometric function, 4-14
TAR, xiii
Technical Assistance Request (TAR), xiii
Terminal symbols
#, A-1
<>, A-2
CONSTRAIN, A-2
definition, A-2
INTEGER_LITERAL, A-2
text
CDL functions, 4-15
TEXT data type, 2-4
text expressions
validation, B-2
text literals
See literals
tokens
definition, 1-3
validation, B-1
trigonometric CDL functions, 4-13
troubleshooting
analyzing errors, xiv
TRUE (Boolean literal keyword), 2-4, 4-22
Truncate
arithmetic function, 4-13

U

unary minus (operator)
description, 4-9
precedence, 4-9
unary plus (operator)
precedence, 4-9
unicode
definition, 1-3
User Properties
validation, B-2

V

validation
Oracle Configurator compiler, B-2
Oracle Configurator parser, B-1
Statement Rules, B-1
variables
local, 4-21
vertical bar (CDL separator)
usage, A-1
violation messages
function overflows and underflows, 4-16

W

WHERE (CDL keyword)
conditional expression, 4-6
example, 3-4
iterator statement, 4-7
whitespace
characters, 4-26
in literals, 4-22
definition, 2-3
rule definitions, 4-25
text literals, 4-22