# DEFENCE RESEARCH ESTABLISHMENT ATLANTIC

# UNIT PACKAGE USER'S GUIDE

RESEARCH AND DEVELOPMENT BRANCH
DEPARTMENT OF NATIONAL DEFENCE
CANADA

D.R.E.A. TECHNICAL MEMORANDUM 80/L

**DEFENCE RESEARCH ESTABLISHMENT ATLANTIC**

9 GROVE STREET
P.O. BOX 1012
DARTMOUTH, N.S.
B2Y 3Z7

TELEPHONE
(902) 426-3100

**CENTRE DE RECHERCHES POUR LA DÉFENSE ATLANTIQUE**

9 GROVE STREET
C.P. 1012
DARTMOUTH, N.É.
B2Y 3Z7

**RESEARCH AND DEVELOPMENT BRANCH**
**DEPARTMENT OF NATIONAL DEFENCE**
**CANADA**

DEFENCE RESEARCH ESTABLISHMENT
ATLANTIC

DARTMOUTH N.S.

D.R.E.A. TECHNICAL MEMORANDUM 80/L *

# UNIT PACKAGE USER'S GUIDE

R. G. Smith  P. Friedland **

December 1980

Approved by R. F. Brown  Director / Underwater Acoustics Division

DISTRIBUTION APPROVED BY

CHIEF D. R. E. A.

RESEARCH AND DEVELOPMENT BRANCH
DEPARTMENT OF NATIONAL DEFENCE
CANADA

* Also published as Stanford Heuristic Programming Project Memo HPP-80-28

** Computer Science Department, Stanford University, Stanford, CA, 94305

## ABSTRACT

The **UNIT Package** is a frame-structured, hierarchically-organized knowledge representation and acquisition system. The package has nodes for individuals, classes, indefinite individuals, and descriptions. Links between the nodes are structured with explicit definitional roles, types of inheritance, defaults, and various datatypes.

This report contains an overview of the **UNIT Package**, a guide to the use of the Unit Editor (**UE**), a summary of the contents of the ***BOOTSTRAP*** knowledge base, and a summary of high-level access functions.

It also contains implementation information, including a discussion of the underlying data structures, global variables and low-level functions,

RÉSUMÉ

L'ensemble UNIT est un système d'acquisition et de représentation d'éléments de connaissance à cadre structuré et organisation hiérarchique. Cet ensemble renferme des noeuds pour les éléments, les classes, les éléments indéfinis et les descriptions. Les liaisons entre les noeuds possèdent une structure découlant de l'utilisation de rôles de définition explicites, de types de relation, de défauts et de divers types de données.

Le présent rapport renferme un aperçu de l'ensemble UNIT, un guide pour l'utilisation du programme d'édition, un résumé de la teneur de la base d'éléments de connaissance BOOTSTRAP (amorçage) et un résumé des fonctions d'accès évoluées.

On y trouve également des renseignements sur la mise en oeuvre, y compris une analyse des structures de données sous-jacentes, des variables globales et des fonctions peu évoluées.

## Table of Contents

# Chapter 1

## Introduction

The **UNIT Package**[1] is a frame-structured, hierarchically-organized knowledge representation and acquisition system. It was originally developed for the **MOLGEN** project at Stanford University **[Stefik, 1979]**, **[Friedland, 1979]**, **[Stefik, 1980]**. The package contains a set of data structures and access functions for program manipulation of those structures. In addition, it contains a sophisticated interactive editor, called **UE**. This editor enables a domain expert (not necessarily a computer specialist) to construct a knowledge base through *direct* interaction with the computer; that is, the transfer of expertise from domain expert to machine need not be mediated by a computer specialist.

This document is intended to serve several purposes and parts of it can be ignored by some readers. The reader is also forewarned that the **UNIT Package** is an evolving system being extended at a number of different sites. As a result, some discrepancies may exist between the system described in this document and the system in use at any given site.

Chapter 2 is an introduction to the basic principles of the **UNIT Package**. Readers who are familiar with these principles may skip this chapter.

Chapter 3 is a user's manual for UE, the Unit Editor. This chapter forms a unit with Chapter 4, which discusses the ***BOOTSTRAP*** knowledge base. This is a basic knowledge base that is intended to form the kernel of *all* user knowledge bases.

Chapter 5 is a detailed discussion of the operation of the access functions offered to a user for program manipulation of UNIT knowledge bases.

Chapter 6 discusses implementation considerations for the **UNIT Package**. It includes a discussion of the underlying data structures. Chapter 7 describes the important global variables. Chapter 8 describes the important low-level internal functions. These chapters need only be read by system implementers and maintainers.

The **UNIT Package** is written in **INTERLISP [Teitelman, 1978]** and runs under the **TENEX** and **TOPS-20** operating systems. One of the implementation features of the package is that it manages the memory residency of the units of a knowledge base. Not all units need be in memory all of the time. When memory is at a premium, the **UNIT Package** moves the slots of some of the memory-resident units to disk. The slots of a unit remain on disk until they must be accessed, whereupon they are returned to memory.

---

[1] The authors wish to acknowledge the contribution of Mark Stefik. He was responsible for much of the original conception of the **UNIT Package** as well as much of the original code.

# Chapter 2

# Basic Principles of the UNIT Package

Knowledge in the UNIT Package is organized as a partitioned semantic network of nodes and links. Following KRL terminology [Bobrow, 1977], the nodes are called units and the links are called slots. Units are connected to each other in a generalization hierarchy with a number of modes of property inheritance. The representation is termed *structured* because it provides a vocabulary of standard node types (for variables and constants), link types (to define properties of the links), and attached procedures to be used uniformly in a knowledge base.

The UNIT Package provides mechanisms for building a knowledge base and operating on its components. It does not, however, provide for a standard interpretation of units. Although units can be created to represent many different entities (e.g., objects, operations, assertions, hypotheses), the interpretation of the units depends on user-supplied software.

## 2.1 Groups: Classifying Units

While a knowledge base is ultimately composed of nodes and links, it is often useful to form larger organizations of units. For this purpose the UNIT Package provides a facility for partitioning a knowledge base into (possibly overlapping) explicit sets (similar to *spaces* in Hendrix's partitioned semantic networks [Hendrix, 1975]). These sets are referred to as ***groups*** in the UNIT Package.

One important use for groups is to denote contexts (e.g., all of the units in a given experimental plan can be put into a group). The UNIT Package provides an efficient notation for groups and a number of functions that operate on all members of a group.

### 2.1.1 UNIT Package Groups

The following is a list of the specific groups recognized and used by the UNIT Package. An explanation of the use of each group is also given.

***DESCR***: Indicates that the unit is a description unit (Section 2.2.3).

***INDEFINITE***: Indicates that the unit is an indefinite unit (Section 2.2.2).

***PRIMITIVE***: Indicates that the unit describes a primitive datatype (Section 2.3.4).

***BOOKKEEPING***: Indicates the unit is used for automatic system documentation and bookkeeping. (Automatic documentation is effected by the interactive editor, UE [Chapter 3]).

***HACKER***: Indicates that the unit will normally only be shown to the user by UE if his profile (Section 3.1) indicates that the he is a *hacker*.

***HIDESLOT***: A classification for datatype units. It indicates that slot whose datatype is so classified will not be shown to the user unless he is a *hacker*. (The effect is analogous to that of a ***HACKER*** classification on a unit.)

***NODELETE***: Indicates that the unit should *never* be deleted automatically, without first asking the user.

## 2.2 Units

The **UNIT Package** provides two basic types of units: instances and specializations. Two other types of units, indefinite units and description units, may be created under special circumstances.

Units are implemented as **INTERLISP** atoms. The information associated with each unit is stored on its property list. (See Chapter 6.)

### 2.2.1 Instances and Specializations

The distinction between instances and specializations may be thought of on two levels: philosophical and practical. Philosophically, that is, thinking of the generalization hierarchy in the **UNIT Package** as a semantic net, instances are used to represent distinct entities; that is, individuals. An instance is analogous to a constant term in predicate calculus. Examples from **MOLGEN** are *Lambda-phage* (a particular DNA molecule), *Plan-step-3* (a particular planning step) or *Electron-microscopy* (a particular laboratory tool).[1]

Specializations are used to represent classes; that is, a branch of the generalization hierarchy.[2] A specialization partially describes all of its potential progeny by expressing all of the attributes that are required for members of the class; it is the generalization of other specializations and instances below it in the hierarchy. Information in a specialization is assumed to be true for all of its progeny and is inherited by them. The specialization *Bacterium* expresses general information about bacteria such as that they have a *gramstain* which may be either positive or negative. A specialization for the bacterial species *E. coli* may inherit this information or add to it as appropriate. In Section 2.3.2 we will see that there are several modes of inheritance and it is necessary to be specific about exactly what is inherited.

Practically, there are only two distinctions made between instances and specializations: only specializations may have progeny (either further specializations or

---

[1] This text uses the convention that all references to specific units and slots are shown in italics. Units are indicated by capitalizing their first letter. For slots, the first letter is not capitalized.

[2] Not to be confused with a ***group*** as described in Section 2.1.

instances) and certain types of slots (see Section 2.3.2) must be given actual values (rather than descriptions of possible values) in instances. Therefore, one may construct a hierarchy entirely of specializations without loss of generality.

### 2.2.2 Indefinite Units

Indefinite units are used to refer to an instance whose identity may be unknown. Different indefinite units can stand for the same individual. They are an augmented form of the existentially quantified variable. The augmentation allows us to indicate that two variables are equal without knowing their values.[3]

Indefinite units are commonly created through the action of the Unit Datatype Editor (see Section 4.3.12). The fundamental links for reasoning about equality for indefinite units in the UNIT Package are (i) an ***ANCHOR*** slot that grounds an indefinite unit to an instance once its identity has been determined, (ii) a ***CO-REFS*** slot that ties indefinite units together, and (iii) a ***NOT-CO-REFS*** slot that separates indefinite units. Two indefinite units are defined to be equal if they have the same anchor or if they are marked as being *co-referential* (i.e., each is contained in the ***CO-REFS*** slot of the other). Two indefinite units are not equal if they have different anchors, or if they are marked as being *not-co-referential* (i.e., each is contained in the ***NOT-CO-REFS*** slot of the other), or if their scopes do not intersect.[4] In all other cases, equality is unknown.

It is often important in planning to indicate that two indefinite objects are the same or different without assigning them to specific constants. In the blocks world, a simple example is when a planner is working to build an arch. It should be able to indicate that the two supporting blocks are *different* before it decides which blocks to use as supports. Without this notational ability, a planner might try to use the same block to support both sides. Indefinite units are also useful for planning in situations involving unknown entities when the entire purpose of the plan is to sort out the identities of objects. The need for such entities in natural language processing (where they are sometimes called *intensional* nodes) has also been recognized [Woods, 1975].

Although the UNIT Package does satisfy the need for a distinguishing and uniform representation of these variable units, numerous difficulties are still left for the user to solve for his individual application. For example, he must provide mechanisms for (i) deciding when there is enough evidence to conclude about equality of entities, and (ii) combining the (possibly conflicting) attributes of the various hypothetical objects once their common identity has been concluded.

---

[3] This is like the augmentation of predicate calculus to predicate calculus with equality except that it provides for the logical possibility that equality may be *unknown*.

[4] A scope in the UNIT Package is simply a branch of the generalization hierarchy.

### 2.2.3 Description Units

***Description units*** are used to refer to a specialization whose identity may be unknown. These units are used for matching and reasoning about abstract classes. Whereas *anchoring* is the key to equality of indefinite units, *matching* is the key to equality of description units. The group defined by a specialization is the complete set of its progeny in the generalization hierarchy; the extension of a description unit is the set of instances and specializations that match it. Two description units are defined to be equal if they potentially match the same units[5] or if they are indicated as being co-referential (i.e., each contains the other in its ***CO-REFS*** slot). Description units are commonly created by the action of the ***MATCH*** command (Section 3.3.13) or the Unit Datatype Editor (Section 4.3.12).

## 2.3 Slots

***Slots*** store the information pertinent to each unit in the hierarchy. They may be thought of as the links between the nodes of a semantic net, or simply as the properties of each class of individual represented by a unit.

Each slot has been given structure through a set of ***fields*** or ***aspects*** associated with the slot. The fields hold information that indicates how a slot is to be interpreted. The **UNIT Package** has a set of predefined fields (called ***permanent fields***) for the name, value, inheritance role, default, and datatype for a slot. The package also supports the addition of fields (called ***optional fields***) for other information, such as attached procedures.

The permanent fields are defined as follows.

***Name***: Name of the slot. All slots are referred to by name.

***Value***: Stored value of the slot. The type of information in the value field is governed by the datatype of the slot and may be influenced by restrictions inherited from its parent in the generalization hierarchy. The **UNIT Package** distinguishes between values that are *terminal* and values that are value-restrictions (i.e., *descriptions* of acceptable values).

***Default***: Terminal value for the slot to be used in the absence of specific information. The contents of the default field must obey the same restrictions as the contents of the value field.

***Datatype***: The datatype field of a slot provides the primary information describing the type of information that can go into the value and default fields of the slot. Examples of datatypes are: links to other units in the hierarchy, integers, strings, lists, and tables. The datatype field is discussed in Section 2.3.4.

***Inheritance Role***: The mode of inheritance of the value field of the slot by progeny of the unit in which it is defined. Four distinct roles are supported by the **UNIT Package** (see Section 2.3.2).

---

[5] The operational test for deciding whether two description units match the same units is that they match each other.

### 2.3.1 Value

The method used to retrieve the value of a slot allows for the case of indirect reference to slots and computed values. If a simple stored value exists in the value field of a slot, then it is returned as the value of the slot. Otherwise, if the slot exists, but has no value, but has a ***TO-GET*** field, then the procedure in that field is invoked. Otherwise, if the slot doesn't exist, but there is a function with the same name as the slot, then that function is invoked with the name of the unit as its argument. Finally, if the slot doesn't exist, but there is a unit with the same name as the slot, then the value finding process commences recursively on the slot named ***VALUE*** in that unit.

### 2.3.2 Inheritance Role

The idea of hierarchical inheritance of properties has its roots in the inferential machinery discussed in Quillian's thesis [Quillian, 1968]. In its simplest form, a value is defined in the most general schema to which it applies. Nodes corresponding to descendants of the schema inherit the value. It has been noted [Brachman, 1977] that, while a slot in an individual is intended to assert a property of the individual, a slot in a schema is often intended to restrict the legal values of corresponding slots in its potential progeny. These alternative meanings of slots (slots used to define properties versus slots used to instantiate properties) are important for the understanding of inheritance and are not always clearly differentiated in network formalisms.

Inheritance roles provide information about the interpretation of the value of a slot. They indicate (i) transmission instructions for the information, and (ii) whether the value is ***criterial*** (fundamental or essential) to the definition of the unit. Inheritance roles are also used by the interactive editor in the UNIT Package to determine what kind of checking is performed when knowledge is entered by the user.

In the UNIT Package the progeny of a unit in the generalization hierarchy *always* inherit slots that are defined in that unit; that is, if a slot with name *color* is defined in unit *A*, then all progeny of *A* will have a slot with name *color*. The default, datatype, and inheritance role fields are also inherited. The value field, however, is treated differently; the inheritance role field of a slot specifies limitations on the contents of the value field and how the contents of the value field are transmitted to progeny of the unit in which the slot is defined. Four inheritance roles (***S***, ***R***, ***O***, and ***U***), are recognized. The four roles are illustrated in Figure 1. The boxes indicate units; the arrows indicate specialization or instance relationships between units.

We will see that the first three roles are decreasingly strict. With with the ***S*** role, a value is directly inherited. With an ***R*** role, direct inheritance does not apply, but a terminal value is required for instances. With the ***O*** role, a terminal value is optional in instances. By contrast, the ***U*** role ignores the nesting of values in progeny altogether.

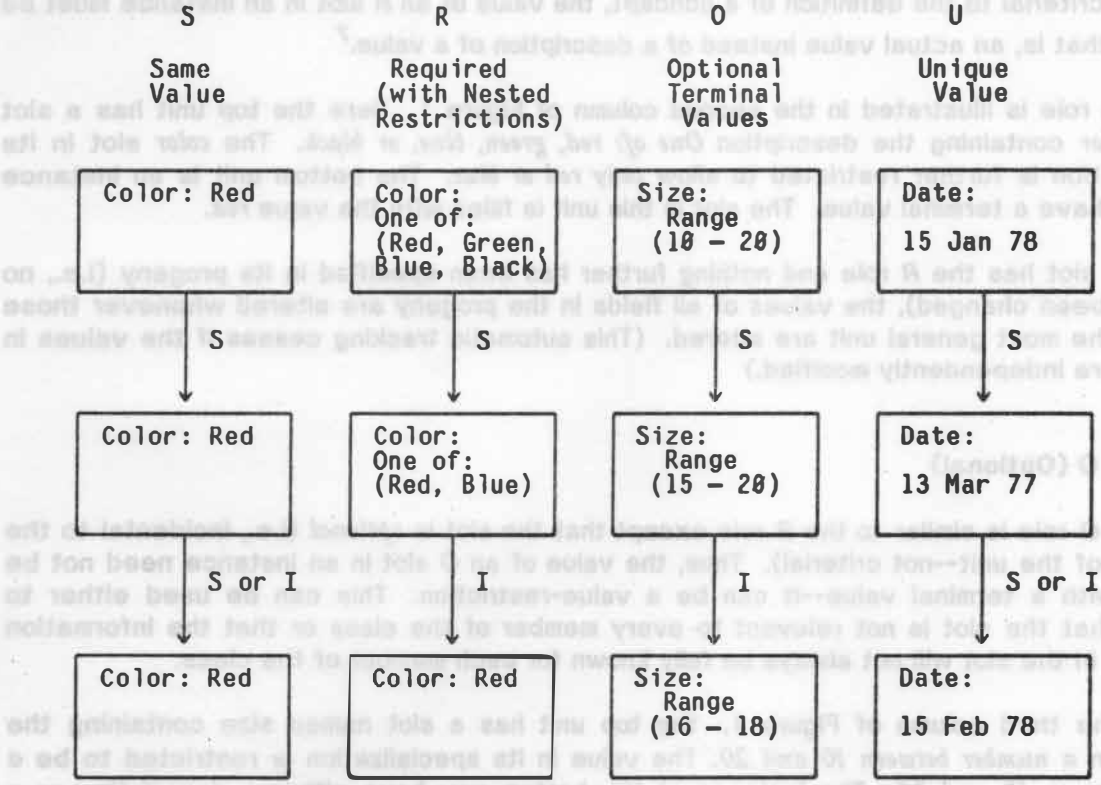


**Figure 1. Modes Of Inheritance.**

#### 2.3.2.1 S (Same)

The simplest role is direct inheritance of slot values by all progeny. This is termed the ***S*** role because the slot has the same value in the defining unit and all of its progeny. An actual value (called a ***terminal value*** as opposed to a restriction) is given for the slot in the value field. Furthermore the **UNIT Package** will only allow that value to be changed in the unit in which the slot was originally defined. This role is illustrated in the leftmost column in Figure 1. All of the units in this column inherit the same value (*blue*) for the slot named *color*.

#### 2.3.2.2 R (Required)

The ***R*** role is for inheritance of *requirements*; it is for slots that are criterial to the definition of a unit. The value in a specialization is interpreted as a restriction on the value fields for the corresponding slots in its progeny. The values in inherited ***R*** slots in progeny

need not be the same as in the specialization, but may be further restricted.[6] Because these slots are criterial to the definition of a concept, the value of an ***R*** slot in an instance must be terminal; that is, an actual value instead of a description of a value.[7]

This role is illustrated in the second column of Figure 1. Here the top unit has a slot named *color* containing the description *One of: red, green, blue, or black*. The *color* slot in its specialization is further restricted to allow only *red or blue*. The bottom unit is an instance and must have a terminal value. The slot in this unit is filled with the value *red*.

If a slot has the ***R*** role and nothing further has been specified in its progeny (i.e., no field has been changed), the values of all fields in the progeny are altered whenever those fields in the most general unit are altered. (This automatic tracking ceases if the values in progeny are independently modified.)

#### 2.3.2.3 O (Optional)

The ***O*** role is similar to the ***R*** role except that the slot is *optional* (i.e., incidental to the definition of the unit--not criterial). Thus, the value of an ***O*** slot in an instance need not be filled in with a terminal value--it can be a value-restriction. This can be used either to indicate that the slot is not relevant to every member of the class or that the information contained in the slot will not always be fully known for each member of the class.

In the third column of Figure 1. the top unit has a slot named size containing the description *a number between 10 and 20*. The value in its specialization is restricted to be *a number between 15 and 20*. The instance at the bottom may have either a description or a number.[8]

If a slot has the ***O*** role and nothing further has been specified in its progeny (i.e., no field has been changed), the values of all fields in the progeny are altered whenever those fields in the most general unit are altered. (This automatic tracking ceases if the values in progeny are independently modified.)

---

[6] The interpretation of *further restricted* is governed by the datatype of the slot. For the ***NUMBER*** datatype, the value-restrictions are numeric ranges or lists. For the ***UNIT*** datatype, the value-restrictions may be description units or ancestor specifications.

[7] The UNIT Package indicates the inheritance role of an ***R*** slot that has been filled with a terminal value as ***(S R)***. This is to show that it behaves the same way as ***S*** role slots with respect to further progeny.

[8] The UNIT Package indicates the inheritance role of an ***O*** slot that has been filled with a terminal value as ***(S O)***. This is to show that it behaves the same way as ***S*** roles slots with respect to further progeny.

#### 2.3.2.4 U (Unique)

The ***U*** role is for information about a node that is not to be inherited by its progeny. The slot value is *unique* at each level in the hierarchy. As with the other roles, the slot name, datatype, and inheritance role are inherited by progeny. However, unlike the other roles, the value is not inherited or used to limit the value of the slot in progeny. The ***U*** role is used mainly for bookkeeping purposes in the **UNIT Package** (e.g., for keeping track of the creator of a unit.)

In the fourth column of Figure 1, a *date* slot (corresponding perhaps to the date that the unit was last changed) is shown in each of the units. The value of the slot in each of the units is a string defining the date, but no relationship is required between the values of the slot in different units.

### 2.3.3 Definitional Role

The inheritance role indicates inheritance paths for properties in the generalization hierarchy. There are, however, other familiar hierarchical relationships that do not indicate inheritance paths. (Some common ones are *element-of* and *part-of*.) Furthermore, not all slots bear the same relation to their parent unit.[9] Consider, for example, some of the slots from the *Organism* unit in a genetics knowledge base (from **[Stefik, 1980]**). The slot *chromosome* is used to refer to a part of the organism; the slot *gramstain* refers to the visible character of the organism when a particular staining agent is applied. Without some way of distinguishing between the different meanings for these slots, a general procedure to separate or remove the parts of a unit might try to remove the gramstain from an organism.

The **UNIT Package** supports distinction of different kinds of slots using a ***definitional role*** that indicates the relationship that the slot bears to its parent unit; that is, it enables the user to specify such a role. This role can be used in the ***DISPLAY*** command to specify a hierarchy for display that is different from the generalization hierarchy (see Section 3.3.6).[10]

The following definitional roles are currently recognized by **UE** (Chapter 3):[11]

***PART-OF***: The slot describes part(s) of the unit.

***SUPER-UNIT***: The inverse of ***PART-OF***. The slot points back to the unit for which the parent unit is a part.

***PROPERTY***: The slot is a property of the unit (an assertion about the unit).

---

[9] This has been recognized by many researchers. Woods, for example, has distinguished between *assertional* and *structural* links **[Woods, 1975]**.

[10] The definitional role is implemented as an optional field (Section 2.3.5) in a slot. (The name of the field is ***DEFN***.)

[11] The extent of the *recognition* is minimal. **UE** recognizes abbreviations for them and lists them as possibilities in response to **?**. None of these roles are currently used anywhere else in the **UNIT Package**.

***RELATION***: The slot describes a binary relation for the unit.

***MANIFESTATION-OF***: The slot describes a manifestation of the unit (e.g., in a passive sonar system, a *line-group* is a manifestation of a platform in the acoustic data).

***DOCUMENTATION***: The slot forms part of the automatic documentation for the unit.

There are of course any number of other possibilities.[12]

### 2.3.4 Datatype

The datatype field of a slot provides the primary information describing the type of information that the value and default fields of a slot can contain. It is similar to the notion of datatypes in conventional programming languages. Hence, a link in the UNIT Package need not go to another unit; it may go to an atom, integer, string, list, or to a value with some other *datatype*.

Each datatype in the UNIT Package is defined as a unit with slots for standard operations.[13] Datatype units are the eventual source of information about how to acquire and edit information that has a specific datatype, how to print values and restrictions of the datatype, how to compare values of the datatype for equality, and so on. This is accomplished by attachment of procedures for these standard operations to datatype units as the values of slots. The procedures are activated when datatype-related operations are requested on the slots. (See Section 2.4.) Factoring datatype-related procedures this way helps to provide a structured approach for adding new datatypes to a knowledge base.

### 2.3.5 Optional Fields

A slot may be annotated with any number of additional fields. A common use of such fields is the attachment of procedures. Optional fields are inherited by progeny in a manner similar to ***U*** slots, except that their values are also inherited. However, the value of an optional field is not subject to any principle of nested restrictions and can be changed in any offspring.

## 2.4 Procedural Attachment

The attachment of procedures to units is one of the ideas common to knowledge representation languages. Several older languages (such as LISP) support arbitrary attachment of procedures to data structures; the approach in knowledge representation languages is more disciplined. This discipline in the UNIT Package has two main elements: (i)

---

[12] The UNIT Package also uses a definitional role called ***EQUIVALENCE*** to label the ***ANCHOR***, ***CO-REFS***, and ***NOT-CO-REFS*** slots of description and indefinite units.

[13] They are all specializations of the unit ***DATATYPE*** in the ***BOOTSTRAP*** knowledge base (Chapter 4).

standardized points for attachment, and (ii) an explicit purpose for each procedure--an indication of when the procedure should be activated. For example, the purpose ***TO-GET*** associated with a procedure that is attached to a slot indicates to the **UNIT Package** that the procedure is to be activated whenever the value of the slot is requested.

The **UNIT Package** provides three standard places for attaching procedures: (i) unit attachment, (ii) slot attachment, and (iii) datatype attachment.[14]

Unit attachment is used for operations that act on the unit as a whole. A possible example is ***IF-DELETED***, which would cause the associated procedure to be activated when a unit was deleted. A procedure is attached to a unit by storing it as the value of a slot whose name is the activation condition (e.g., ***IF-DELETED***).[15]

Slot attachment is used for operations on a particular slot of a unit (e.g., ***TO-GET***, described above). A procedure is attached to a slot by storing it as the value of a field whose name is the activation condition (e.g., ***TO-GET***).

Datatype attachment is used for operations on slots located throughout the knowledge base that have values of a particular datatype (e.g., ***PRINT*** for value printing). A procedure is attached to a datatype by storing it as the value of a slot (of the datatype unit) whose name is the activation condition (e.g., ***PRINT***). Although datatype attachment has not been reported in other knowledge representation systems, it has been used extensively in the **UNIT Package**. Similar ideas were used in **SIMULA [Dahl, 1966]**.

### 2.4.1 Messages For Procedure Activation

Some terminology has been adapted from **SMALLTALK [Goldberg, 1976]** to describe the activation of attached procedures. A procedure is activated by sending it a *message* with a token that matches its purpose. This is an indirect form of procedure call--where the purpose of the procedure is known to the caller but the name of the procedure need not be known.

There are two basic forms of message: unit message and slot message. A ***unit message*** is sent to a unit. The procedure that is the value of the slot whose name is the *purpose* token of the message is invoked. If no such procedure exists, then an error is generated.

A ***slot message*** is sent to a slot. In this case, the process is potentially more complex. First the **UNIT Package** looks for the procedure in the field of the slot whose name is the *purpose* token. If there is no field of that name in the slot, then the package checks the datatype unit for the slot. It looks for a procedure as the value of a slot whose name is the *purpose* token. If there is still no procedure, then the **UNIT Package** looks for one in a unit whose name is the slot name. In this case it looks for a procedure as the value of a slot whose name is the *purpose* token. If no procedure can be found, then an error is generated.

---

[14] Datatype attachment may be interpreted as a special case of unit attachment because datatypes are represented as units in a **UNIT Package** knowledge base.

[15] No such procedures are currently implemented in the **UNIT Package**.

# Chapter 3

# UE: The Unit Package Knowledge Base Editor

The Unit Editor, UE, provides a user interface to the *UNIT Package*. Under normal circumstances, a user should never have to use any INTERLISP functions to create and maintain a UNIT knowledge base.[1]

UE provides facilities for creating, examining, and modifying knowledge bases, as well as facilities for combining information from a number of different knowledge bases. It is designed to be almost self-documenting and to provide aid to inexperienced users while at the same time not bothering experienced users with over-prompting.

The idea of using an interactive editor for knowledge base construction and management stems from a desire to ease the problem of transfer of expertise from a domain expert to a program. The intent is to enable the domain expert to construct a knowledge base by himself, without the necessity of directly involving a computer scientist. There is evidence that this can both speed the expert-to-program transfer of expertise and improve its accuracy [Friedland, 1979]. A speed improvement is possible when non-experts do not have to be intimately involved in describing each object in the knowledge base. An accuracy improvement is possible when a computer scientist intermediary can be bypassed. The computer scientist is not necessarily an expert in the problem domain. When the knowledge is complex and subtle, and its purpose may not be immediately apparent to a non-expert, some distortion must inevitably result.

It is not clear that experts who are not computer specialists will always be able to logically formalize their knowledge of a domain. One of the most useful roles played by the computer scientist intermediary is to help provide a logical organization for the knowledge of an expert. It is no doubt true, however, that the expert-to-program transfer of expertise is easier if the expert can examine and understand the knowledge base as well as a computer scientist can.

## 3.1 Getting Started with UE

A UNIT knowledge base consists of at least two files: the UNITS file and the RELATIONS file. The UNITS file contains all of the slot information for the units in the knowledge base. It is used as an extended memory by the UNIT Package memory management procedures (see Section 6.2). The RELATIONS file holds all other information about the units in the knowledge base (the immediate ancestor and progeny in the generalization hierarchy, the groups to which the unit belongs, etc.). A third file may also be used; the HASH file. This file contains entries for slots with datatype *TEXT* (see Section 4.3.11).

---

[1] Accessing values in the knowledge base during planning or other use will require use of the functions described in Chapter 5.

To get started, run UE from the UNITS system directory at your site (e.g., <AI> at DREA). UE will ask you for the name of the knowledge base you wish to edit. You will be given the choice of making a copy of the knowledge base to protect you from destroying it through error.

You are allowed at this point to build a new knowledge base, either by starting with a copy of an existing knowledge base, or by starting fresh. In most cases you will want to start with the ***BOOTSTRAP*** knowledge base. It contains a number of primitive and bookkeeping units that help to make the **UNIT Package** usable. Typing ***?*** or ***HELP*** or ***SHOW*** will give you assistance at this juncture.

You will then be given the chance to load any additional LISP functions that may be special to your knowledge base. ***UE*** will try to find compiled files if you specify no file extension for your LISP functions.

You will then be given the UE prompt of "UE:." At this point if you type ***?*** or ***HELP*** you will get a list of allowable commands. Type ***!HELP*** for a brief explanation of the commands (see Appendix A). These commands are explained in detail in Section 3.3.

### 3.1.1 Interaction With UE: General Information

The principle of ready assistance without bothering the experienced user with continual unwanted advice controls all interaction with UE. The system prompts the user with a symbol, word, or phrase whenever it is expecting input. The user may type ***?*** or ***HELP*** at any time for a list of allowable responses or response types, or in some instances may type ***HELP*** for more detailed assistance. Flexibility is provided by permitting users to type ahead instructions whenever they know what information the system will be requesting. If the requested operation(s) can be completed without error, then the user will not be bothered with unnecessary prompting.

Commands are generally recognized by the minimal set of characters that makes them unambiguous with respect to other possible commands. For example, ***CR***, ***CRE***, ***CREA***, ***CREAT***, and ***CREATE*** will all invoke the ***CREATE*** command. Command names are also completed in some versions of ***UE*** when ***<escape>*** or a delimiter (e.g., ***<space>***, ***<return>***) is typed. You may use ***?*** to determine possible alternatives at any point. In addition, unit and slot names are subjected to spelling correction and the system will complete unit and slot names that are terminated with ***<escape>*** as long as this can be done unambiguously.[2] UE makes no distinction between small and capital letters.

The problem of providing for both top-down and bottom-up construction of a knowledge base has been difficult to achieve in a conceptually *clean* fashion. The major problem is that if the user, operating in a top-down fashion, wishes to fill a slot with a pointer to another unit that he has not yet defined, the normal consistency checking mechanism of the **UNIT Package** cannot be applied. To facilitate operation in a top-down mode, a recursive feature has been added. At any point, the user can make a recursive call to the top-level of UE (see

[2] For some critical commands you will be asked to approve the correction.

Section 3.2.1). He may then define the needed unit, return via the normal exit procedure, and fill in the slot as desired. This recursion may be carried out to as many levels as desired and UE informs the user which level he is in upon request and entry to a new level and return to a previous level.

UE performs basic automatic documentation functions. It keeps track of the dates of creation and last modification, as well as the names of the creator and last modifier. Every unit also has a slot for a text description.

The UNIT Package *always* checks for violations before carrying out any operation and prints an error message in case of trouble. The error messages include an error number, a message, and an indication of the unit, slot, field, relation, function, role, group, or file thought to be in error (or a list of several of these). The complete list of error messages is shown in Appendix C.

## 3.2 The User Profile

Another facility that allows UE to provide for different levels of user is the ***user profile***. The system has built-in facilities for controlling the amount of information provided during certain interactions. For example, slots can be described in either a brief mode, where only the value of the slot is printed, or a verbose mode where datatype and inheritance role information is also given. The mode chosen is dependent on the user profile.

The user profile is a small file stored on the user's directory, called UNITS.PROFILE. If this file exists, then it is used to control the settings of a number of flags. The package also allows a user to override the profile settings or generate a new profile (see the ***SET-PROFILE*** command [Section 3.3.24]).

### 3.2.1 Control Characters

At any point in an editing session four special control characters are operational:

***↑K*** will recursively invoke UE at the top-level. For example, if you were creating a unit and found that another unit you wanted to use did not yet exist, you could type ***↑K***, create that unit, type *OK* and be back at the point you were when you had the problem. The system will remind you of how deep you are in the recursion whenever you type ***↑K*** or *OK*.

***↑E*** at any time in a subcommand will abort that subcommand and get you back to the point at which that subcommand was given. (CAUTION: Only type ***↑E*** to a prompt. Otherwise, you may compromise the state of your knowledge base.)

***↑P*** at any time will show you where you are in UE. It will tell you the recursion level and the name of the knowledge base. It will also show you basically what you are doing (e.g., what function you are in), in order of increasing scope. The following is an example.

```
You are at recursion level 1
You are editing Knowledge Base TEMP-CIRCUIT under TOPS20 on DREA
    Function: EDITATOM
    Function: SLOTMSG
    Editing Slot: B
    Editing Unit: JONES
    Network Editor
    Network Editor
```

*(This appears twice because the user is at recursion level 1, and not the top-level.)*

*↑O* at any time will stop the current output.

If for any reason you are returned to **INTERLISP** (e.g., if you inadvertently type *↑D*), type ***(UE-TOP)*** to come back to **UE**.

## 3.3 UE Commands

### 3.3.1 COMPACT

The ***COMPACT*** command allows you to unify the free blocks on your UNITS file (see Section 6.2). It also allows you to expand the space allocated to each unit to allow for future growth without reorganization of the data structure (by a factor between 1.0 and 5.0--a reasonable factor is 1.1).

### 3.3.2 CONSISTENCY

The ***CONSISTENCY*** command allows you to verify the consistency of the value information in the slots of units in your knowledge base. You will be asked for the name of a unit at which to start checking consistency. All units below this unit in the generalization hierarchy will be checked. You will then be asked for the names of groups to which to restrict the checking.

### 3.3.3 COPY

The ***COPY*** command allows you to copy a unit in the currently specified network to a new unit. You will be asked for the parent of the new unit. You will also be asked for the relation of the copy to the parent (depending on your profile--see Section 3.3.24).

### 3.3.4 CREATE

The ***CREATE*** command allows you to create a new unit in the knowledge base. It will ask you for the new name, a parent unit, and whether it is an ***INSTANCE*** or a ***SPECIALIZATION*** (depending on your profile--see Section 3.3.24). It will then

automatically fill ***BOOKKEEPING*** slots the unit inherits from its parent (things like creation date, description, etc.) asking you to fill slots that require your interaction (e.g., the ***DESCR*** slot). If the unit is an instance, then you will also be asked to fill in terminal values for any ***R*** slots that it inherits.

You will then be told that you can add new slots or edit existing slots with the ***EDIT*** command (Section 3.3.8).

The prompt "CREATE:" indicates that the ***CREATE*** command is waiting for input.

### 3.3.5 DELETE

The ***DELETE*** command allows you to delete a unit in the knowledge base. You will be asked whether the progeny of the unit are also to be deleted.

If you choose not to delete the progeny, then all of the progeny of the unit become progeny of the parent of the unit (i.e., a level is removed from the generalization hierarchy).

Notes: Any top-level slots in the deleted unit (i.e., slots that are defined in that unit) will disappear from all of the progeny when the unit is deleted. In addition, the ***SHOWREFS*** command (Section 3.3.25) is available to check for references to a unit in the slots of other units before it is deleted. This helps to eliminate possible inconsistencies in the knowledge base. It is, however, a *slow* process because it entails an exhaustive search of the knowledge base.)

### 3.3.6 DISPLAY

The ***DISPLAY*** command allows you to get an overall view of the knowledge base. Normally, it prints the names of units in the generalization hierarchy. You will be asked to name a unit at which to start the display and whether you wish to display all units (***BOTH***--this is the default), just ***SPECIALIZATIONS***, just ***INSTANCES***, or just members of specific ***GROUPS*** (in which case you will be asked for the name(s) of the group(s)). You will also be allowed to specify a print level for the display; that is, the number of levels of the hierarchy to be displayed.

You are also able to depart from the generalization hierarchy with the ***DISPLAY*** command. You can specify that the display should go through particular slots, either by naming the slots, or by naming definitional roles (i.e., the display will go through slots with certain names, or slots that have certain definitional roles). To do this, use ***SLOTS*** or ***DEFINITIONS*** in place of the options mentioned earlier.

The name of each unit is displayed with its progeny below it--indentation indicating level in the hierarchy. Instances are displayed with an asterisk before and after. Description units are followed by "(Descr)" and indefinite units are followed by "(Indef)." When you display a group, all units are followed by "←." The display is sensitive to your profile (see Section 3.3.24).

The prompt "DISPLAY:" indicates that the ***DISPLAY*** command is waiting for input.

### 3.3.7 DONE

The ***DONE*** command tells UE that you are finished for this session. You will be asked whether the knowledge base is to be saved on disk. If you respond affirmatively, the knowledge base is saved and you are returned to the operating system. Otherwise you are asked whether you wish to cancel the session. If you respond affirmatively, then the latest versions of the UNITS, RELATIONS, and HASH files are deleted (i.e., the versions that were made when you started UE) and you are returned to the operating system. Otherwise, you are returned to INTERLISP with the current knowledge base still open.

*OK* is a synonym for ***DONE***.[3]

### 3.3.8 EDIT

The ***EDIT*** command starts the slot editor. This editor controls the fundamental operation of knowledge acquisition, that of manipulating slots within units. It allows you to examine and modify units in the knowledge base. It will ask you what unit you wish to modify. The prompt "EDIT:" indicates that you are dealing with the slot editor. The current slot editor commands are described in the following sections. Type ***HELP*** for a brief explanation of the commands (see Appendix B).

#### 3.3.8.1 CLASSIFY

The ***CLASSIFY*** command allows you to examine, add, and remove classifications associated with the unit (i.e., groups in which the unit is a member). The prompt "CLASSIFY:" means that the ***CLASSIFY*** command is waiting for input. The possible options are: ***ADD***, ***DELETE***, ***PRINT***, and ***DONE*** (*OK*). ***ADD*** (***DELETE***) will prompt you for the classification to be added (deleted). You will also be given the option of adding (deleting) the classifications to (from) the progeny of the unit.

#### 3.3.8.2 CDEFAULT

The ***CDEFAULT*** command clears the default value field of a slot. You will be asked for the name of the slot and whether the default field is to be cleared for the corresponding slot in the progeny of the unit.

---

[3] This is true almost universally in UE. The exceptions occur when dealing with INTERLISP itself (e.g., returning to UE from INTERLISP must be done with ***OK--DONE*** will not work).

#### 3.3.8.3 CLEARVALUE

The *CLEARVALUE* command clears the value field of a slot. You will be asked for the name of the slot. You will also be asked whether the value is to be cleared for the corresponding slot in the progeny of the unit.

#### 3.3.8.4 COPY

The *COPY* command allows you to copy the value of a slot from another unit to the one currently being edited. It will ask you for the name of the slot in the current unit which is to get the copy (you are allowed to create a new slot at this point) and then for the slot name and unit name of the slot of which a copy is to be made. *COPY* does not copy the default and optional fields of the slot.

#### 3.3.8.5 CREATE

The *CREATE* command allows you to add a new slot to the current unit. You will be asked for the new slot name (it has to be unique in the unit), the datatype of the slot, the inheritance role of the slot, and the definitional role (if you have specified that option in your profile [see Section 3.3.24]). You will then be put into the editor for the datatype you specified. (See Section 4.3 for descriptions of individual datatype editors associated with datatypes in the *BOOTSTRAP* knowledge base.) Checks will be made to ensure that the value you give for the slot is consistent with the inheritance role (e.g., if you gave an inheritance role of *S* you cannot give a restriction as a value).

#### 3.3.8.6 DELETE

The *DELETE* command allows you to delete a slot in the unit. (The slot must have been defined in the current unit.) You will be asked for the name of the slot you wish to delete. The slot will also be deleted in all progeny of the unit.

#### 3.3.8.7 DISPLAY

The *DISPLAY* command lists the names of the slots in the current unit. The display is dependent on your profile (see Section 3.3.24).

#### 3.3.8.8 DONE

The *DONE* command releases you from the slot editor and returns you to the top level of UE. (*OK* is a synonym for *DONE*.)

### 3.3.8.9 EDIT

The ***EDIT*** command allows you to edit an existing slot. You will be asked for the name of the slot you wish to edit and then you will be transferred to the datatype editor for the datatype of that slot (see below). The slot editor makes sure that the changes you make are consistent with the datatype of the slot, with the inheritance role of the slot, and with its inherited restrictions. (You cannot, of course, change the value of ***S*** slots that have been inherited, or of ***R*** or ***O*** slots that have been given terminal values in a higher unit in the generalization hierarchy.)

One further series of command options can result during use of the ***EDIT*** command. If you put a value into a slot that causes conflicts from above or below in the generalization hierarchy--you violate a restriction inherited from above, or make a new restriction that invalidates values stored below the unit--you will be given the option to change the slot value for the unit you are editing or to change the conflicting values from above or below.

As a result of the changes you make, certain description or indefinite units may become extraneous. The slot editor keeps track of any description or indefinite units that were pointed at when you entered the datatype editor. If they are not pointed at when you exit, the slot editor will ask if you wish them to be deleted. Some care must be exercised here since the slot editor only keeps track of the fact that the slot just edited no longer points at these units. They may, however, still be referenced by other slots in the current unit, or in some other unit.

#### Individual Datatype Editors

In order to edit a slot value you are transferred to the datatype editor associated with the datatype of the slot. A variety of datatypes has been created for the ***BOOTSTRAP*** knowledge base (Chapter 4). Each of these datatypes has a specialized editor (Section 4.3). These datatypes have been provided to get the user started. Because datatypes are generally domain-specific, he will more than likely wish to augment the set discussed in this document. The procedure for doing this is discussed in Section 4.2. The user should be forewarned that this entails a fair amount of effort. He must write procedures for editing, printing, and so on.

### 3.3.8.10 FIELD-EDIT

The ***FIELD-EDIT*** command enables you to add, delete, or edit optional fields of a slot (i.e., fields other than value, default, inheritance role, and datatype). You will be asked for the name of the slot whose fields you wish to edit. The prompt "fEDIT:" indicates that the Field Editor is waiting for input.

The following field edit commands are available: ***CREATE***, ***DELETE***, ***EDIT***, ***OK*** (***DONE***), and ***PRINT***. In all cases, you will be asked for the name of the field. ***CREATE*** allows you to simply create a field. (You will not be asked to enter a value). You may give a field a value with ***EDIT***. (You may specify the name of new field to ***EDIT***. The new field will be created and you will be asked for a value.) When a field is given a value, you are given the option of

propagating it to the progeny of the unit. ***DELETE*** allows you to delete a field. ***PRINT*** allows you to print a field. (***PRINT*** can be followed with ***ALL*** to print *all* optional fields.)

#### 3.3.8.11 !HELP

The ***!HELP*** command prints a list of slot editor commands with terse explanations (see Appendix B). This listing can be aborted by typing ***^O***.

#### 3.3.8.12 MSG

The ***MSG*** command enables you to send a message to a field of a slot. You will be asked for the name of the slot and the token (i.e., the field name). When you enter the field name, you may follow it with other arguments. For example, typing: ***MSG FOO THINK JOE*** sends a *think* message to slot *foo*; i.e., activates the procedure in the *think* field of the slot or in the *think* slot of the associated datatype with the argument ***JOE***. The prompt "SMSG:" indicates that the ***MSG*** command is waiting for input.

#### 3.3.8.13 OK

The ***OK*** command releases you from the slot editor and returns you to the top level of UE. (***DONE*** is a synonym for ***OK***.)

#### 3.3.8.14 PRINT

The ***PRINT*** command allows you to print some or all of the slots in the current unit. You will be given a "Print:" prompt. Type ***<return>*** or ***ALL*** to print all of the slots. Type ***T*** to print all top-level slots. Type ***S, U, R, O*** to print all slots of that inheritance role. Type ***E*** to print all slots except ***BOOKKEEPING*** slots. Type ***NEW*** to print all slots with new values (i.e., slots that have been defined by an ancestor of the unit but whose value has been defined in the unit). Type ***<name>*** to print a slot of that name.[4]

If your user profile has been set to ***VERBOSE*** (see Section 3.3.24), then the ***PRINT*** command will print the name of each slot, the inheritance role, the datatype, the name of the unit in which it was originally defined, or "*Top*" if it is a toplevel slot, the value, and the names and values of all optional fields.

If your profile is set to ***BRIEF***, then only the slot name and value are printed. The default field is printed only if a default value has been set and the value in the value field is not terminal or your profile is set for ***DEFAULT***. Not all slots are shown unless your profile indicates that you are a *hacker*.

---

[4] If the unit has a slot whose name corresponds to one of the above keywords, then that keyword will not be operational.

#### 3.3.8.15 !PRINT

The ***!PRINT*** command is the same as the ***PRINT*** command except that your profile is temporarily set to ***VERBOSE HACKER NO-WITH-NOTATION*** for a complete printout. (See Section 3.3.24.)

#### 3.3.8.16 RENAME

The ***RENAME*** command allows you to rename a slot in the unit. You will be asked for the old slot name and the new slot name. You can only make such a change for top-level slots. The name will also be changed in the progeny of the unit.

#### 3.3.8.17 SETDEFAULT

The ***SETDEFAULT*** command allows you to set a default value for a slot. You will be asked for the name of the slot and transferred to the the datatype editor for the datatype of the slot. The slot editor will check to ensure that the value you give is consistent with the inheritance role and restrictions for the value of the slot, and is a valid terminal value for the datatype of the slot. (If you have specified a *value-restriction* as the value for the slot, then the default must obey that restriction; otherwise, it must meet any inherited restrictions.) You will be given the option of propagating the default value to the progeny of the unit.

#### 3.3.8.18 SHOWRELATIONS

The ***SHOWRELATIONS*** command displays the units that are related to the current unit in the generalization hierarchy as ***GEN***eralization, ***PROTO***type, ***SPEC***ialization, and ***INST***ance.

#### 3.3.8.19 SORT

The ***SORT*** command sorts the slots in a unit. The possible criteria are: ***A***, ***H***, ***R***, or ***O***.

***A***: The slots are placed in *alphabetical* order.

***H***: The slots are placed in *hierarchical* order. This means that slots are ordered according to the level in the generalization hierarchy from which they have been inherited. Slots inherited from the highest unit in the generalization hierarchy appear first. Slots inherited from the same unit are placed in alphabetical order.

***R***: The slots are placed in *reverse hierarchical* order.

***O***: The slots are ordered by the *user*. If you specify ***O***, you will be told the current order of the slots and asked to specify a new order, one slot at a time. You may escape at any time by typing ***DONE*** or ***OK***, in which case the remaining slots will be left in their original order.

You are prompted each time with the name of the current next slot. Type ***<return>*** to accept it, or type the name of another slot.

### 3.3.9 GROUP

The ***GROUP*** command allows you to perform operations on groups of units (see Section 2.1). The current ***GROUP*** operations are: ***COPY***, ***CREATE***, ***DELETE***, ***DISPLAY***, ***OK*** (***DONE***), ***PRINT***, and ***REMOVE***.[5]

#### 3.3.9.1 COPY

The ***COPY*** command allows you to copy a group. You will be asked for the name of the group to be copied, and the name of the new group. New units will be created for each of the units in the group to be copied, and the old units will be copied to them. Inter-unit references will be fixed in the new group so that they refer to the new units.

#### 3.3.9.2 CREATE

The ***CREATE*** command allows you to create a group of existing units. You will be asked for the name of the group and the units that are to be its members. If the group already exists, then you may add new units to it.

#### 3.3.9.3 DELETE

The ***DELETE*** command allows you to delete all units in a group. You are asked for the name of the group. You are then given the choice of deleting all units in the group and their progeny, or just the prototype units. Finally, you are asked whether or not you wish to confirm deletion of individual units.

If you have chosen not to delete all units of the group, then the slots of the remaining units in the group are checked for references to units that no longer exist. References of this type are cleared. (NOTE: This process only holds for the ***UNIT*** and ***LIST*** datatypes in the ***BOOTSTRAP*** knowledge base.)

#### 3.3.9.4 DISPLAY

The ***DISPLAY*** command allows you to display the names of all known groups.

---

[5] Manipulation of some of the groups in the ***BOOTSTRAP*** knowledge base is only allowed when your profile indicates that you are a *hacker* (see Section 3.3.24).

#### 3.3.9.5 DONE or OK

***DONE*** or ***OK*** returns you to UE.

#### 3.3.9.6 PRINT

The ***PRINT*** command allows you to print the names of the units in a group. You are asked for the name of the group.

#### 3.3.9.7 REMOVE

The ***REMOVE*** command allows you to remove a group classification. You are asked for the name of the group. It is removed from the classification list for all of its units.

### 3.3.10 !HELP

The ***!HELP*** command prints a list of UE commands with terse explanations (see Appendix A). This listing can be aborted by typing ***↑O***.

### 3.3.11 INSTANTIATE

The ***INSTANTIATE*** command changes a specialization to an instance. You are asked for the name of the unit. Instantiation is only possible if the unit has no progeny and has no ***R*** slots without terminal values. (You can always use the slot editor (Section 3.3.8) to give these slots terminal values before using the ***INSTANTIATE*** command.)

### 3.3.12 LISP

The ***LISP*** command transfers you to INTERLISP. OK returns you to UE. (This is one place where OK and DONE are not synonyms. In addition, OK must be in upper-case, because it is typed at INTERLISP instead of UE.)

NOTE: You should be aware that automatic memory management (Section 6.2) is normally in effect while you are interacting with the UNIT Package. This means that you must be especially careful about the use of ***↑E***. Its use at an unfortunate time could result in a trashed knowledge base. However, this control character is often essential while functions are being debugged. Therefore, in order to enable its safe use, a facility has been incorporated to disable memory management. This is done by setting the global variable ***UA.PFLG*** to NIL. Reset it to T when you wish memory management to be reinstated.

### 3.3.13 MATCH

The ***MATCH*** command prints a list of units that match a description. You are asked for the name of the parent unit (i.e., the unit to use as the prototype for the units you want to match). You are then asked whether you want to search all progeny of the prototype without regard to group, or whether you want to limit the search to progeny in particular groups.

A description unit is created by this command to hold the description for the match. You will then be asked to fill in the slots of the description and will be transferred to the slot editor. (You may add slots as well as modifying the existing slots defined in the prototype.)

A match is deemed to occur when the slot values specified in the description unit are consistent with the restrictions of the unit being matched. (Optional fields are ignored for matching purposes.)

For example, you can say ***MATCH ORGANISM***. This indicates that you want to return all progeny of *Organism* that meet certain criteria. These criteria are specified by filling in the slots of a description unit whose prototype is *Organism*. For example, if *slot1* of *Organism* can be filled with either RED or BLUE, and we set its value to RED in the description unit, then ***MATCH*** will return all progeny of *Organism* for whom *slot1* = RED.

### 3.3.14 MOVE

The ***MOVE*** command enables you to move a unit from one place in the generalization hierarchy to another. You will be asked for a new parent and given the option of moving the progeny of the unit.

If any conflicts exist between slots in the unit to be moved and slots of the new parent, you will be informed. Slots that are inherited from the new parent take precedence over slots with the same name in the unit to be moved (or its progeny). The offending slots in the unit will not be lost. They will be copied into top-level slots with names of the form "$$-<slotname>."

If you do not move the progeny, then all instances are deleted, and specializations become specializations of the old generalization of the unit. Top-level slots in the unit are deleted in its old specializations.

### 3.3.15 MSG

The ***MSG*** command enables you to send a message to a slot of a unit. You will be asked for the name of the unit and the token (i.e., the slot name). When you enter the slot name, you may follow it with other arguments. For example, typing: ***MSG FOO THINK JOE*** sends a *think* message to unit *Foo*; i.e., activates the procedure in the *think* slot of *Foo* with the argument ***JOE***.

### 3.3.16 ?MSGS

The *?MSGS* command lists the tokens for procedures in a unit (i.e., the names of slots that can respond to messages sent to the unit). You are asked for the name of the unit.

### 3.3.17 NETWORK

The *NETWORK* command allows you to change the knowledge base you are working on. It will close the current knowledge base and ask you for the name of the next knowledge base you wish to work on. The dialogue is similar to that encountered when you start UE.

### 3.3.18 OK

The *OK* command tells UE that you are finished for this session. You will be asked whether the knowledge base is to be saved on disk. If you respond affirmatively, the knowledge base is saved and you are returned to the operating system. Otherwise you are asked whether you wish to cancel the session. If you respond affirmatively, then the latest versions of the UNITS, RELATIONS, and HASH files are deleted (i.e. the versions that were made when you started UE) and you are returned to the operating system. Otherwise, you are returned to INTERLISP with the current knowledge base still open.

*DONE* is a synonym for *OK*.[6]

### 3.3.19 PRINT

The *PRINT* command lets you see a single unit in detail. You are asked for the name of the unit and the print options. Type *<return>* or *ALL* to print All slots. Type *T* to print all top-level slots. Type *S*, *U*, *R*, *O* to print all slots of that inheritance role. Type *E* to print all slots except *BOOKKEEPING* slots. Type *NEW* to print all slots with new values (i.e., slots that have been defined by an ancestor of the unit but whose value has been defined in the unit). Type *<name>* to print a slot of that name.[7]

*PRINT* will show the group classifications (if any) of the unit and what unit is its generalization or prototype. If your user profile has been set to *VERBOSE* (see Section 3.3.24) then the name of each slot, the inheritance role, the datatype, the name of the unit in which it was originally defined, or "*Top*" if it is a toplevel slot, the value, and the names and values of all optional fields are printed. If your profile is set to *BRIEF*, then only the slot name and value are printed. The default field is printed only if a default value has been set and the value in the value field is not terminal or your profile is set to *DEFAULT*. Not all slots are shown unless your profile indicates that you are a *hacker*.

---

[6] This is true almost universally in UE. The exceptions occur when dealing with INTERLISP itself (e.g., returning to UE from INTERLISP must be done with *OK--DONE* will not work).

[7] If the unit has a slot whose name corresponds to one of the above keywords, then that keyword will not be operational.

### 3.3.20 !PRINT

The *!PRINT* command is the same as the *PRINT* command except that your profile is temporarily set to *VERBOSE HACKER NO-WITH-NOTATION* for a complete printout. (See Section 3.3.24.)

### 3.3.21 RECORD

The *RECORD* command enables you to make a recording of this session in a specified file. For example, REC FOO.BAR makes a recording in file FOO.BAR. A subsequent *REC* command stops the recording.

### 3.3.22 RENAME

The *RENAME* command enables you to rename a unit. You are asked for the present name of the unit and the new name. You are then asked if all references to the old unit in the slots of other units are to be renamed as well. In any case, description and indefinite units are correctly renamed and their *DESCR* slots are modified to reflect the change. (NOTE: Checking references is a *slow* process. It necessitates a search through the entire knowledge base.)

### 3.3.23 SAVE

The *SAVE* command causes the current knowledge base UNITS, RELATIONS, and HASH files to be copied. It then returns you to UE.

### 3.3.24 SET-PROFILE

The *SET-PROFILE* command allows you to modify various UE flags that control some UNIT Package operations. You will be prompted for a profile command. Current profile commands are shown below. For each command that sets a flag, its opposite is shown in parentheses and its effect is shown underneath.

*ASK-XMIT (NO-XMIT)*

You will be asked if you want values of slots that are changed to be propagated to all progeny, including those for which the slots have more specific values. If you specify no transmission, you will still be informed if a new value causes conflicts and asked to fix them (with help).

*BRIEF (VERBOSE)*

When printing slots, print only the slot name, the value, and the optional fields. The alternative is to print the slot name, inheritance role, datatype, the name of the unit in which it was defined (or "*Top*" if it is a top-level slot), the value, and the optional fields.

***CRT (NO-CRT)***

This setting may depend on your site. At some sites, ***CRT*** enables line-editing of all input, ***NO-CRT*** disables it. At others, ***CRT*** sets up backspace for a display terminal, ***NO-CRT*** sets up backspace for a hardcopy terminal. Check with your local **UNIT Package** programmer.

***DEFAULT (NO-DEFAULT)***

Always print the default field. The opposite is to print the default field only if a default value has been set and the value in the value field is not terminal.

***DEFN (NO-DEFN)***

Always prompt for a definitional role during slot creation. The opposite is to never prompt.

**DONE** or ***OK***

Return to UE.

***HACKER (USER)***

Enable printing of all ***BOOKKEEPING*** and ***PRIMITIVE*** units and all slots whose datatype units are members of the ***HIDESLOT*** group by the ***DISPLAY*** and ***PRINT*** commands. The opposite is to disable this printing.

***NOISY (QUIET)***

UE will inform you about units and attached procedures that are transferred by the ***TRANSFER*** command. The opposite is to make the transfers silently.

**PRINT** or ***SHOW***

Print your current profile.

**STORE**

Store your current profile on a file called UNITS.PROFILE on your directory. This file is loaded each time you use the **UNIT Package.**

***WITH (NO-WITH)***

Use KRL-like *with* notation for printing description and and indefinite units; that is, print the description in the following form: *a Block with front = a panel with color = blue* for description units, or *some Block with front = a panel with color = blue* for indefinite units. The opposite is to print only the name of the description or indefinite unit (e.g., ***$X-1***).

### 3.3.25 SHOWREFS

The ***SHOWREFS*** command shows all references to a unit in the slots of other units in the knowledge base. You are asked for the name of the unit and whether the search is to be restricted to units in particular groups. ***SHOWREFS*** also notes references to any description and indefinite units that were created from the specified unit. (Note that this command entails an exhaustive search of the knowledge base and is slow.)

### 3.3.26 SPECIALIZE

The ***SPECIALIZE*** command changes an instance into a specialization.

### 3.3.27 SPLITUNIT

The ***SPLITUNIT*** command allows you to take an existing unit and split it into parent and child, giving you the option of which slots go into parent and which into child. You will be asked to give the name of the unit to be split--this will become the child of the parent-child pair--and the name to be given to the new unit. You will then be asked to specify if all top-level slots are to move to the new parent (indicated by ***A***), if none are to move (***N***), or if you want to select which top-level slots are to move (***O***). If you choose the latter option, ***SPLITUNIT*** will go through all of the unit's slots and allow you to choose which ones to move. (Non-top-level slots automatically appear in the new unit.)

### 3.3.28 SUMMARYFILE

The ***SUMMARYFILE*** command allows you store a text summary of the current knowledge base on a disk file. It will ask you for a unit with which to start (like the ***DISPLAY*** command) and will produce a file with the selected units in the same format as that produced by the ***PRINT*** command. You will also be asked for the number of days back to include a ***WHATSNEW*** type report of changes to the knowledge base (see Section 3.3.31). (A response of ***0*** indicates that no report is desired.) In addition, you may specify that only specific groups of units are to be shown. Finally, you are given the option of printing only slots that originate or are changed in each unit.

Both the slots shown and the format in which they are shown are dependent upon your profile (see Section 3.3.24).

### 3.3.29 TRANSFER

The ***TRANSFER*** command allows you to move individual units from another knowledge base into your current knowledge base. You are asked for the name of the other knowledge base, the unit to be transferred, the unit in the current knowledge base that is to be its parent (only if the parent of the unit does not exist in the current knowledge base), and whether you wish the progeny of the unit to be transferred as well.

If the unit already exists in the current knowledge base, then the normal procedure is to add new slots and group classifications, but not to alter existing slots and group classifications. This can lead to trouble when slots with the same name are defined differently. You are therefore offered the option of remaking the unit (this cannot be done for datatypes since they are required to effect the actual transfer).

Units in the slots of the unit you wish to transfer are also transferred. ***TRANSFER*** also keeps track of units and procedures transferred. (This prevents the same unit from being transferred more than once if it is pointed to by several units that are transferred.) If you transfer a unit with attached procedures, the procedures must be transferred by hand.

Unless ***QUIET*** has been specified in your profile (see Section 3.3.24), ***TRANSFER*** will inform you of all transfers.

### 3.3.30 TSHOW

The ***TSHOW*** command shows units and attached procedures that have been transferred from other knowledge bases to the current knowledge base, along with the names of the knowledge bases from which they were transferred. If several units or attached procedures with the same name are transferred from various knowledge bases, they are only shown as being associated with the last knowledge base from which they were transferred.

### 3.3.31 WHATSNEW

The ***WHATSNEW*** command allows you to see what changes have been made recently to a unit and all of its progeny in the current knowledge base. It will ask how many days back it should check for changes and and the name of the unit at which to start the examination (all progeny are also checked).

# Chapter 4

# BOOTSTRAP: An Initial Knowledge Base

ROOT
  DATATYPE
    ATOM
    BOOLEAN
    EXPR
    INTEGER
    INTERVAL
    LISP
    LIST
    NUMBER
    STRING
    TABLE
    TEXT
    UNIT

  CREATED
  MODIFIED
  CREATOR
  MODIFIER
  DESCR

**Figure 2. The BOOTSTRAP Knowledge Base.**

There are a number of units of such wide generality that almost anyone who builds a UNIT knowledge base will want to have these units included. To provide for this need, an initial knowledge base of very general units has been prepared. It is called ***BOOTSTRAP*** and resides on the UNITS system directory This knowledge base contains a number of primitive datatypes as well as units for automatic documentation. For any particular application, a user can augment the set of datatype units.

## 4.1 The Basic Units

### 4.1.1 ROOT

***ROOT*** is the top-level unit of any knowledge base. In the ***BOOTSTRAP*** knowledge base, this unit contains the following documentation slots. These slots are then inherited by all other units.

| SLOT | ROLE | DATATYPE |
|---|---|---|
| DESCR | (U) | <DESCR> |
| MODIFIER | (U) | <MODIFIER> |
| CREATOR | (U) | <CREATOR> |
| MODIFIED | (U) | <MODIFIED> |
| CREATED | (U) | <CREATED> |

### 4.1.2 DATATYPE

***DATATYPE*** is the generalization of units used as the datatypes in a knowledge base. Every slot in every unit in the knowledge base has a datatype which must be one of these units. Section 4.2 explains the addition of new datatypes to a knowledge base.

***DATATYPE*** is a member of the ***HACKER*** and ***PRIMITIVE*** groups. It has the following slots.

| SLOT | ROLE | DATATYPE |
|---|---|---|
| COPY | (U) | <LISP> |
| DELETE | (U) | <LISP> |
| EDIT | (U) | <LISP> |
| EQUALS | (U) | <LISP> |
| PRINT | (U) | <LISP> |
| STRICTER | (U) | <LISP> |
| TERMINALVALUE | (U) | <LISP> |
| TEST | (U) | <LISP> |
| TRANSFER | (U) | <LISP> |
| UNITS | (U) | <LISP> |

Every datatype (specialization of the ***DATATYPE*** unit) will respond to the messages: ***COPY***, ***DELETE***, ***EDIT***, ***EQUALS***, ***PRINT***, ***STRICTER***, ***TERMINALVALUE***, ***TEST***, and ***TRANSFER***. The function of these slots and the arguments that are passed to the attached procedures are discussed below.

In the following text, whenever we refer to a ***datatype instance***, we mean either a terminal value or a value restriction with a particular datatype.

***COPY***: Invoke a procedure to copy an instance of the datatype. This is only necessary for datatypes for which the INTERLISP function COPYALL is not sufficient (e.g., ***LIST*** and ***UNIT***). The arguments are shown below.

Arg1: Datatype instance to copy
Arg2: Slot
Arg3: Unit

***DELETE***: Invoke a procedure to delete an instance of the datatype. This is only necessary for datatypes such that an **INTERLISP** deletion is not sufficient (e.g., ***LIST*** and ***UNIT***). The arguments are shown below.

Arg1: Datatype instance to delete
Arg2: T if instance is to be deleted without user interaction
Arg3: Slot
Arg4: Unit
Arg5: T if instance is not really contained in Slot of Unit

***EDIT***: Invoke an editor to create an instance of the datatype. ***EDIT*** procedures are expected to simply return the new value for the slot. The **UNIT Package** automatically checks for consistency and deposits the value in the slot. This convention of not actually depositing the value in the slot allows the datatype editors to be used in situations where their editing expertise is required but the value is targeted for some storage place other than a slot in a unit, for example, as part of some composite data structure. For those cases where it is desirable that the datatype editor perform this checking, it can return the special token ***NOTEST*** to inhibit this activity in the **UNIT Package**. The arguments are shown below.

Arg1: Datatype instance to edit
Arg2: Restriction on value in Slot of Unit
Arg3: Slot
Arg4: Unit

***EQUALS***: Invoke a routine to decide if two instances of the datatype are equal. The arguments are shown below. Unless NOTESTFLG is T, both Arg1 and Arg2 will be checked to make sure that they are valid datatype instances. (If type checking fails, then ***EQUALS*** procedures return NIL.)

Arg1: Datatype instance 1 (value in Slot of Unit)
Arg2: Datatype instance 2 (comparison value)
Arg3: Slot
Arg4: Unit
Arg5: NOTESTFLG (T if type checking is not to be done)

***PRINT***: Print an instance of the datatype. The arguments are shown below.

Arg1: Datatype instance to be printed
Arg2: NOCRFLG (If T, do not terminate with <return>)
Arg3: Slot
Arg4: Unit
Arg5: POSLST (List of fallback character positions (see Section 5.7.5)

***STRICTER***: Invoke a procedure to decide whether an instance of a datatype satisfies a restriction or whether a valid restriction for an instance of the datatype is more strict than another restriction. Its arguments are shown below. ***STRICTER*** functions are expected to return NIL if the restriction is met and a complaint string (which can be printed to explain the violation) in case of error. ***STRICTER*** functions must obey the following rule:

*Restriction A is STRICTER than Restriction B if and only if The set of Terminal Values STRICTER than Restriction A is contained in the set of Terminal Values STRICTER than Restriction B.*

In cases where a dummy restriction procedure is desired that does no checking, the routine ***NILTEST*** is included in the UNIT Package. It always returns NIL.

Arg1: Datatype Instance
Arg2: Restriction on value in Slot of Unit
Arg3: Slot
Arg4: Unit

***TERMINALVALUE***: Invoke a routine to decide if an instance of the datatype is a terminal value. ***TERMINALVALUE*** routines are expected to return T if the value is a terminal value, else NIL. In those cases where a dummy procedure is desired, the routine ***NOTEST*** is included in the UNIT Package. It always returns T.

Arg1: Datatype instance
Arg2: Not used
Arg3: Slot
Arg4: Unit

***TEST***: Invoke a routine to decide if a structure is a valid instance of the datatype.

Arg1: Datatype instance
Arg2: Not used
Arg3: Slot
Arg4: Unit

***TRANSFER***: Invoke a routine to decide if the transfer of an instance of the datatype (by the ***TRANSFER*** command [Section 3.3.29]) implies that other units should also be transferred. ***TRANSFER*** routines are expected to return a list of units that should be transferred together with a flag (T or NIL) for each unit that indicates whether its progeny should also be transferred. The list is of the form ( ... ($Unit_1$ $Flag_1$) ... ). The arguments are shown below.

Arg1: Datatype instance
Arg2: Not used
Arg3: Slot
Arg4: Unit

***UNITS***: Invoke a routine to return a list of the units mentioned in the instance of the datatype.

Arg1: Datatype instance
Arg2: Not used
Arg3: Slot
Arg4: Unit

### 4.1.3 Individual Datatypes

All of the individual datatype units are specializations of ***DATATYPE***. They are all members of the ***HACKER*** and ***PRIMITIVE*** groups.

#### 4.1.3.1 ATOM

***ATOM*** is the datatype corresponding to INTERLISP atoms. Value restrictions are allowed in the form of lists of atoms.

#### 4.1.3.2 BOOLEAN

***BOOLEAN*** has three possible values: True, False, and Unknown. No value restrictions are allowed. In INTERLISP this corresponds to the literal atoms T, F, and NIL.

#### 4.1.3.3 EXPR

***EXPR*** is the datatype for INTERLISP expressions. Value restrictions are allowed. At present they are simply expressions that have been marked as value restrictions. An ***EXPR*** is implemented as an INTERLISP list. The first element is T if the expression is a terminal value, NIL if a value restriction. The actual value of the ***EXPR*** is given by CAADR of the list.

#### 4.1.3.4 INTEGER

***INTEGER*** is the datatype for integers. Value restrictions are allowed in the form of lists of integers or ranges of integers. In INTERLISP, an ***INTEGER*** is implemented as either a normal integer or a list of integers, or a list with R as the first element. Positive and negative infinity are represented by the largest positive and negative integers possible on the DEC-10 or DEC-20.

#### 4.1.3.5 INTERVAL

***INTERVAL*** is the datatype for closed intervals bounded by two numbers. One-sided intervals are also allowed. (The *open* side boundary is filled in with plus or minus infinity, as appropriate.) Value restrictions are allowed in the form of subintervals, where terminal values must be a subset of the interval specified.

An ***INTERVAL*** is implemented as an INTERLISP list of three elements. The first element is T if the interval is a terminal value, NIL if a value restriction. The remaining elements are floating point numbers or integers. The second element is the lower bound. The third element is the upper bound. Positive and negative infinity are represented by the largest positive and negative floating point numbers possible on the DEC-10 or DEC-20.

#### 4.1.3.6 LISP

***LISP*** is the datatype for INTERLISP functions. No value restrictions are allowed. ***LISP*** has an additional ***PP*** slot (shown below) that contains a function for **PRETTYPRINT**ing an INTERLISP function.

| SLOT | ROLE | DATATYPE |
|---|---|---|
| PP | (U) | <LISP> |

#### 4.1.3.7 LIST

***LIST*** is the datatype for lists. The list elements must all have the same datatype (which can also be ***LIST***). Value restrictions are allowed in that each of the list elements can be a valid value restriction for the datatype of the list. ***LIST*** has the additional slots shown below. ***GET-DATATYPE*** contains a procedure for returning the datatype of the elements of the list. ***GET-ELEMENT*** contains a procedure for returning a particular element of the list. ***GET-LIST*** contains a procedure for returning the list in INTERLISP form (i.e., **UNIT Package** implementation tokens are stripped off). ***PUT-LIST*** contains a procedure for replacing a list in a slot. (See Section 5.10 for more detailed discussion of these procedures.) A ***LIST*** is represented as an INTERLISP list of the form:
(TOKEN DATATYPE ELEMENT$_1$ ELEMENT$_2$ ... ELEMENT$_n$).
***DATATYPE*** is the datatype of the individual elements. ***TOKEN*** indicates how the list is to be interpreted (as a terminal value or value-restriction). The following token forms are recognized.

***<integer>***: Terminal value lists that satisfy this value-restriction must have <integer> elements.

*L* or ***(L <integer>)***: The list represents a terminal value. (The individual elements must *all* be terminal values in this case.)

*S* or ***(S <integer>)*** or ***ONE*** or ***(ONE <integer>)***: The list represents a value-restriction. The individual elements may be either terminal values or value-restrictions. The interpretation is that every element in a terminal value list must be ***STRICTER*** (according to the datatype of the list) than *some* restriction element of this list.

***ALL*** or ***(ALL <integer>)***: The list represents a value-restriction. The individual elements may be either terminal values or value-restrictions. The interpretation is that that every restriction element in this list must be satisfied (according to STRICTER for the datatype of the list) by *some* element of a terminal value list.

The actual value of the ***LIST*** is given by CDDR of the list.

| SLOT | ROLE | DATATYPE |
|---|---|---|
| GET-DATATYPE | (U) | <LISP> |
| GET-ELEMENT | (U) | <LISP> |
| GET-LIST | (U) | <LISP> |
| PUT-LIST | (U) | <LISP> |

#### 4.1.3.8 NUMBER

***NUMBER*** is the datatype for real numbers. Value restrictions take the form of lists of numbers or ranges of numbers. In INTERLISP, a ***NUMBER*** is implemented as either a normal floating point number (or integer) or a list of numbers, or a list with R as the first element. Positive and negative infinity are represented by the largest positive and negative floating point numbers possible on the DEC-10 or DEC-20.

#### 4.1.3.9 STRING

***STRING*** is the datatype for strings. Value restrictions take the form of lists of strings.

#### 4.1.3.10 TABLE

***TABLE*** is the datatype for tables. All values in the same column must have the same datatype. The column names are arbitrary labels. Value restrictions are constructed by defining the column datatypes and labels, but not the column values.

#### 4.1.3.11 TEXT

***TEXT*** is the datatype for long discourse. No value restrictions are allowed.

#### 4.1.3.12 UNIT

***UNIT*** is the datatype used for units. Valid terminal values are unit names. Value restrictions take several forms: One can restrict a slot value to be one of a list of units, one of the progeny of a unit, an instance of a unit, a unit matching a specified description, an indefinite unit, a unit mentioned by another unit, or a unit found in some slot of another unit. (See Section 4.3.12 for more detailed discussion.)

#### 4.1.3.13 BOOKKEEPING DATATYPES

***CREATED***, ***MODIFIED***, ***CREATOR***, ***MODIFIER***, and ***DESCR*** are all special datatypes used for automatic documentation. They are all members of the ***BOOKKEEPING*** group.

***CREATED*** stores the creation date for a unit (as a string). It is also a member of the ***HIDESLOT*** group. The ***EDIT*** slot contains a procedure that gets the day's date and time by looking at the system clock.

MODIFIED stores the modification date of a unit (as a string). It is also a member of the ***HIDESLOT*** group.

***CREATOR*** stores the creator of a unit (as a string). It is also a meber of the ***HIDESLOT*** group. The ***EDIT*** slot contains a procedure that gets the name of the creator.

MODIFIER stores the name of the modifier of a unit (as a string). It is also a member of the ***HIDESLOT*** group.

***DESCR*** is used for text description slots.

All of the ***BOOKKEEPING*** datatypes have the following slot.

| SLOT | ROLE | DATATYPE |
|---|---|---|
| USER-INTERACTION | (U) | <BOOLEAN> |

If the value of this slot is T, then the user will be prompted for a value for the associated datatype instance when it is created.

### 4.2 Creating New Datatypes

To create a new datatype, do the following:

Create a new unit that is a specialization of ***DATATYPE***. Then give values for the ***LISP*** datatype slots: ***COPY***, ***DELETE***, ***EDIT***, ***EQUALS***, ***PRINT***, ***STRICTER***, ***TERMINALVALUE***, ***TEST***, and ***TRANSFER*** (some of these can be ignored--examine the primitives in the ***BOOTSTRAP*** knowledge base for details).

### 4.3 The Individual Datatype Editors

All of the primitive Datatypes in the ***BOOTSTRAP*** knowledge base have their own editors that allow you to give terminal values or restrictions (depending on the role) for instancess of those datatypes.

To exit from all of the datatype editors give the command ***DONE*** (or ***OK***). If the value given is legal (i.e., is a valid datatype instance and meets all restrictions inherited from

above and currently imposed from progeny) you will move on to your next step of UE interaction. Otherwise you will be told what is wrong and be given the chance to re-edit the value.

### 4.3.1 The ATOM Editor

The ATOM Editor indicates its presence by the prompt "AE:"

Current ATOM Editor commands are:

| | |
|---|---|
| ADD | Add New Atom |
| DELETE | Delete Existing Atom |
| HACKER | Invoke LISP EDITV (For Hackers) |
| PRINT | Print Value |
| SHOW | Show Inherited Restrictions |
| STOP or ABORT | Abort the Edit |
| UNDO | Undo Previous Command |
| CLEAR | Clear Value |
| OK or DONE | Done |

You may also type a single atom as the value, or a list of atoms as a value-restriction. When entering a single atom that could be mistakenly recognized as a command, precede the atom with "!".

### 4.3.2 The BOOLEAN Editor

The BOOLEAN Editor indicates its presence by the prompt "BE:"

Current BOOLEAN Editor commands are:

| | |
|---|---|
| TRUE | Set Value to True |
| FALSE | Set Value to False |
| PRINT | Print Value |
| CLEAR | Clear Value |
| OK or DONE | Done |

If no value is given, the value will be taken as a value-restriction allowing true or false.

### 4.3.3 The EXPR Editor

The EXPR Editor indicates its presence by the prompt "EE:"

Current EXPR Editor commands are:

| | |
|---|---|
| HACKER or EDIT | Invoke LISP EDITV (For Hackers) |
| PRINT | Print Expression |
| RESTRICTION | Expression is a Value-Restriction |
| TERMINAL | Expression is a Terminal Value |
| SHOW | Show Inherited Restrictions |
| CLEAR | Clear Value |
| OK or DONE | Done |

The first time you invoke EDITV, you will notice that the "value" in the edit buffer is (NIL). This simply makes it possible to call EDITV before a real value has been set. Ignore the outer parentheses, and reset the NIL to whatever value you wish.

### 4.3.4 The INTEGER Editor

The INTEGER Editor indicates its presence by the prompt "IE:"

Current INTEGER Editor commands are:

| | |
|---|---|
| LIST | Enter a List of Integers as a Value-Restriction |
| RANGE | Enter a Range of Integers as a Value-Restriction |
| PRINT | Print Value |
| SHOW | Show Inherited Restrictions |
| CLEAR | Clear Value |
| OK or DONE | Done |

You may also give a single integer as a terminal value. For the RANGE command, -I is recognized as -INFINITY, and +I is recognized as +INFINITY.

### 4.3.5 The INTERVAL Editor

The INTERVAL Editor indicates its presence by the prompt "InE:"

Current INTERVAL Editor commands are:

| | |
|---|---|
| SUBINTERVAL n1 n2 | Enter a Subinterval as a Value-Restriction |
| PRINT | Print Value |
| SHOW | Show Inherited Restrictions |
| CLEAR | Clear Value |
| OK or DONE | Done |

You may also type n1 n2 as a terminal value. For the SUBINTERVAL command, -I is recognized as -INFINITY, and +I is recognized as +INFINITY.

### 4.3.6 The LISP Datatype Editor

The LISP Editor indicates its presence by the prompt "LispE:"

Current LISP Editor commands are:

| | |
|---|---|
| EDIT | Invoke LISP EDITF to edit the function (For Hackers) |
| PRINT | Print Value |
| STOP or ABORT | Abort the Edit |
| CLEAR | Clear Value |
| OK or Done | Done |

You may also type the NAME of the function. If the function name could be mistakenly recognized as one of the commands, precede it with "!".

If you type the name of an already existing function it will be taken as the value of the slot. Otherwise you will be asked if you wish to create a function of that name and be put into EDITF with a skeletal function of the name given.

### 4.3.7 The LIST Editor

For constructing lists, a distinction is made between *simple* datatypes for which terminal values can be given by a single string (e.g., ***ATOM***, or ***STRING***), and *complex* datatypes which require a datatype editor to be accessed (e.g., ***LIST***).

The LIST Editor indicates its presence by the "LE:" prompt.

Current LIST Editor commands are:

| | |
|---|---|
| ADD e1 ... en | Add entries e1 ... en to the list. (They must be terminal values of *simple* datatypes.) |
| DATATYPE d | Specify the list will contain Datatype d |
| DELETE m | Remove mth entry from the list |
| EDIT | Add a new entry with the Datatype Editor |
| EDIT m | Edit mth entry with Datatype Editor. |
| HACKER | Invoke LISP EDITV (For Hackers) |
| TERMINAL or LIST | List is a Terminal Value (This is the default.) |
| REPLACE e1 ... en | Replace the current list entries with e1 ... en (They must be terminal values of a *simple* datatype.) |

| | |
|---|---|
| RESTRICTION or SUBLIST or ONE | List is a Value-Restriction. Some of the entries must be satisfied in a terminal value. (This is the default.) |
| ALL | List is a Value-Restriction. All entries must be satisfied. (A terminal list will match this list if every item in the terminal list matches at least one item in this list.) |
| PRINT | Print Value |
| SHOW | Show Inherited Restrictions |
| CLEAR | Clear Value |
| OK or DONE | Done |

You may also type an integer to specify how many items the list must have (0 clears the current length specification).

### 4.3.8 The NUMBER Editor

The NUMBER Editor indicates its presence by the "NE:" prompt.

Current NUMBER Editor commands are:

| | |
|---|---|
| RANGE n1 n2 | Enter a Range as a Value-Restriction |
| LIST | Enter a List of Numbers as Possible Values |
| PRINT | Print Value |
| SHOW | Show Inherited Restrictions |
| CLEAR | Clear Value |
| OK or DONE | Done |

You may also give a single number as a value. For the RANGE command, -I is recognized as -INFINITY, and +I is recognized as +INFINITY.

### 4.3.9 The STRING Editor

The STRING Editor indicates its presence by the prompt "SE:".

Current STRING Editor commands are:

| | |
|---|---|
| ADD | Add New String |
| DELETE | Delete Existing String |
| HACKER | Invoke LISP EDITV (For Hackers) |
| PRINT | Print Value |
| SHOW | Show Inherited Restrictions |
| STOP or ABORT | Abort the Edit |
| UNDO | Undo Previous Command |
| CLEAR | Clear Value |
| OK or DONE | Done |

You may also type a single string as the value, or a sequence of strings (separated by spaces) as a value-restriction. (Quote marks are optional unless a string contains spaces.) When entering a single string that could be mistakenly recognized as a command, precede the string with "!".

### 4.3.10 The TABLE Editor

For constructing tables, a distinction is made between *simple* datatypes for which terminal values can be given by a single string (e.g., ***ATOM***, or ***STRING***), and *complex* datatypes which require a datatype editor to be accessed (e.g., ***LIST***).

The TABLE Editor indicates its presence by the "TE:" prompt.

Current TABLE editor commands are:

| | |
|---|---|
| ADD l1 d1 ... ln dn | Add new columns with specified labels and datatypes (*simple* datatypes) |
| CHANGE n l d | Respecify the label and datatype of column n |
| COLUMN i v1 ... vn | Replace column i entries with v1 ... vn |
| DEFINE l1 d1 ... ln dn | Define a table with specified column labels and datatypes |
| ENTRY m i v | Replace a single entry in row m and column i (must be a terminal value of a *simple* datatype) |
| EDIT m i | Replace a single entry in row m and column i with an entry generated with a datatype editor |
| HACKER | Invoke LISP EDITV (For Hackers) |
| REPLACE m v1 ... vn | Replace row m entries with v1 ... vn (must be terminal values of a *simple* datatype) |
| ROW v1 ... vn | Fill in a row of the table with entries v1 ... vn (must be terminal values of a *simple* datatype) |
| PRINT | Print Value |
| CLEAR | Clear Value |
| OK or DONE | Done |

### 4.3.11 The TEXT Editor

The TEXT Editor indicates its presence by the prompt "te:"

Current TEXT Editor commands are:

| | |
|---|---|
| EDIT e | Invoke System Text Editor (one of EMACS, VMACS, TV, or SOS). EDIT with no argument invokes same editor as last time. |
| HASH | Convert string text to hashfile entry |
| STRING | Convert hashfile entry to string text |
| PRINT | Print Value |
| CLEAR | Clear Value |
| OK or DONE | Done |

Or you may simply type text to the "te:" prompt. Each line of text will be concatenated to the text that was entered previously. To go back and fix a previous line, you will need to use the EDIT command. If the first word could be mistakenly recognized as a command, precede it with "!". To insert an arbitrary string, surround the input line by "". If you do not do this, then atoms are all that will be accepted; that is, tabs, parentheses, groups of spaces, and so on will be ignored. (This will not prevent command recognition from being attempted.)

In order to conserve space in memory, a hashfile facility has been added to the UNIT Package. This facility enables you to store ***TEXT*** datatype slots on a hashfile. You are still able to edit these entries, just as if they were represented as strings, but a distinct savings in string space is realized. To convert an existing string ***TEXT*** entry to a hashfile entry, use the ***HASH*** command. (If no hashfile exists, then one will be created.)

### 4.3.12 The UNIT Datatype Editor

The UNIT Editor indicates its presence by the prompt "UnE:".

Current UNIT Editor commands are:

CLEAR: Clear the current value.

SHOW: Show Inherited Restrictions.

EDIT: Edit the current value (if it is a unit).

!HELP: Print an expanded summary of the commands.

LIST: Specify that the unit is one of a list (e.g. *Chevy Ford Cadillac Jaguar*). The members of the list must be terminal unit names (i.e., they cannot be descriptions or restrictions).

NONE: No unit. Used to emphasize absence of a link.

OK or DONE: Done

PRINT: Print the current value.

*I: Specify that the value is an instance of a given unit or of one of its offspring (e.g. **I Vehicle*).

*P: Specify that the value is an offspring (instance or specialization at some level) of a given unit (e.g., **P Vehicle*).

*IND: Specify a value that is an indefinite unit--a particular instance of some as yet undetermined unit (e.g., *$car-33*--the car I saw). An indefinite unit is created (i.e., a unit that is a member of the ***INDEFINITE*** group). You are asked to name a prototype unit and are then placed in the Slot Editor to fill in the slot values for a match. Three extra slots are added (each with definitional role ***EQUIVALENCE***): ***ANCHOR***, ***CO-REFS***, and ***NOT-CO-REFS***. The ***ANCHOR*** slot is filled if this unit is ever equated with an instance. Its datatype is ***UNIT*** and its initial value is NONE. The ***CO-REFS*** slot is filled if this unit is equated to another indefinite unit. It has datatype ***LIST*** and has the initial value "Some of: UNITs:." The ***NOT-CO-REFS*** slot is filled if this unit determined to be not-co-referential to another indefinite unit. It has datatype ***LIST*** and has the initial value "Some of: UNITs:."

*D: Specify a value by describing the components of a unit (e.g., **D Vehicle*). A description unit (i.e., a unit that is a member of the ***DESCR*** group) is created. You are asked to name a prototype unit and are then placed in the Slot Editor to fill in the slot values for a match. Two extra slots are added (with definitional role ***EQUIVALENCE***): ***CO-REFS*** and ***NOT-CO-REFS***. These slots are filled if this unit is determined to be co-referential or not-co-referential to another unit. They have datatype ***LIST*** and initial value "Some of: UNITs:." Unlike indefinite units, description units are never *anchored* to a particular instance and do not have an ***ANCHOR*** slot. They are mostly used for matching. Description units are defined to be equal if they are co-referential or if they match. Descriptions can be printed in KRL-like ***WITH*** notation (e.g., *A Bacterium with gramstain = negative*). (This depends on your profile. See Section 3.3.24.)

*R: Specify an indirect reference to a unit by specifying an access path (e.g., *Ref to chromosome of Organism*). This is useful when the method to get the name of the unit can be given, but the unit name is not known (or the unit is not yet defined). The reference may be (i) to the value of a slot in some other unit or (ii) to some unique unit in a group. Hence, you are given the option of referring to a particular slot. You are then asked to specify a unit and given the option of making the reference to one of its progeny instead of to the unit itself. In this case, you are given the option of limiting the search to units in a particular group.

*M: Specify an indirect reference by naming a unit that mentions it and specifying the ancestor of the unit mentioned. This supports indirect references like *The Culture mentioned in Lab-Goal-3*. It differs from the pathname references above in that (i) no slot is specified in which to find the reference and (ii) an ancestor of the referent unit (e.g., *Culture*) is specified. Hence, you are asked to name the ancestor of the unit that is mentioned, and the unit in which the unit of interest is mentioned. You are also given the option of specifying a particular group of units.

You may also specify a unit by name. When entering a single unit that could be mistakenly recognized as a command, precede the unit with "!".

# Chapter 5

## Functions In The UNIT Package

You may not always want to interact with UE to make alterations to a knowledge base. The following functions are useful for manipulation of a UNIT knowledge base under program control.

### 5.1 Functions That Operate On Knowledge Bases

#### 5.1.1 NETWORK? (FILENAME FLG)

NETWORK? returns information about knowledge base FILENAME according to the value of FLG as follows:

CURRENT: T if FILENAME is the current knowledge base.

EXISTS: T if a knowledge base with name FILENAME exists.

FOREIGN: T if FILENAME contains a directory name and the directory name does not correspond to that of current user.

MINE: Performs a directory search on the connected directory. Returns a list of the knowledge bases that are found.

ANY: Performs a directory search through all of the accounts on the global list ***UA.USERS***. Returns a list of the names of the knowledge bases that are found (with directory names).

NIL: The name of the current knowledge base.

#### 5.1.2 OPENNETWORK (FILE FLG)

OPENNETWORK initializes the UNIT Package for the knowledge base named FILE. FILE is expected to be an atom corresponding to the name field of a TENEX/TOPS-20 file. (It should not include any extension or version number.) A knowledge base consists of two files (plus an optional third file):

***<FILE>.UNITS***: The UNITS file contains all of the slot information for the units in the knowledge base. It is used as an extended memory by the UNIT Package memory management procedures (Section 6.2).

***<FILE>.RELATIONS***: The RELATIONS file holds all other information about the units in the knowledge base (the immediate ancestor and progeny in the generalization hierarchy, the groups in which the unit is a member, the address of the slot information in the UNITS file, the relative time at which the slots of the unit were accessed, and a flag that indicates whether the unit has been modified since the last time it was written to disk).

***<FILE>.HASH***: The HASH file holds entries for the values of slots with datatype ***TEXT***. (This file is optional.)

If FLG = NOFNS, then the functions for the knowledge base are not loaded. This is useful for situations where it is known that the functions have already been loaded (e.g., after a call to COPYNETWORK when a knowledge base is copied from another directory). If a compiled version of the functions is available, it is loaded. Otherwise the source file of the functions is loaded.

OPENNETWORK has no effect if the current knowledge base is open. If a knowledge base is successfully opened, OPENNETWORK enables unit memory management and returns the name of the knowledge base. Otherwise, it returns NIL.

### 5.1.3 CLOSENETWORK (FLG)

CLOSENETWORK writes the current knowledge base out to disk. It has no effect if there is no current knowledge base. In this case, it returns NIL. CLOSENETWORK normally writes out the UNITS, RELATIONS, and HASH files, closes them, undefines the units in the current knowledge base, disables unit memory management and returns T. If FLG = T, CLOSENETWORK makes copies of the UNITS, RELATIONS and HASH files, then returns ready for further operations on the current knowledge base. In this case, its value is the name of the current knowledge base.

### 5.1.4 CANCELNETWORK ()

CANCELNETWORK cancels the current session. Operationally, this involves closing the UNITS file and then deleting the most recent versions of the UNITS, RELATIONS, and HASH files. This deletes all changes to the current network back to the previous CLOSENETWORK. CANCELNETWORK also disables unit memory management. CANCELNETWORK has no effect if no network is open. It returns T if it is successful, else NIL.

### 5.1.5 MAKENETWORK (FILE)

MAKENETWORK initializes the unit access functions for a new (and almost empty) knowledge base in memory. It also enables unit memory management. FILE specifies the name of the knowledge base. MAKENETWORK has no effect if a knowledge base is already open. If successful, MAKENETWORK returns the name of the current knowledge base, else NIL. The new current knowledge base is defined with one unit--the ***ROOT*** unit, with no slots defined.

**NOTE: Usually, the appropriate way to create a new knowledge base is to start with a copy of the *BOOTSTRAP* knowledge base as noted earlier.**

### 5.1.6 COPYNETWORK (FROMFILE TOFILE)

COPYNETWORK makes a complete copy of all the relevant files for one version of the knowledge base. If both FROMFILE and TOFILE are specified, a new version of the TOFILE knowledge base equivalent to the FROMFILE knowledge base will be created (i.e., the UNITS, RELATIONS, and HASH files). If TOFILE = NIL, a new version of the FROMFILE knowledge base is created. If FROMFILE is from another user's directory and TOFILE = NIL, a knowledge base of the same name will be created in the current user's directory. COPYNETWORK returns the name of the knowledge base if successful, otherwise NIL. The HASH file is actually rehashed and not simply copied. This helps to remove accumulated garbage that may result from editing text entries.

### 5.1.7 COMPACT (FACTOR)

COMPACT unifies the free blocks on the UNITS file (see Section 6.2) and expands the space allocated to each unit to allow for future growth (by a FACTOR between 1.0 and 5.0--a reasonable factor is 1.1). COMPACT also rewrites the RELATIONS file to account for the new disk addresses on the UNITS file. COMPACT returns the size of the free block on the UNITS file (in bytes).

## 5.2 Functions That Operate On Units:

### 5.2.1 UNIT? (UNIT)

UNIT? returns UNIT if it is the name of a unit in the current knowledge base and NIL otherwise.

### 5.2.2 MAKEUNIT (UNIT RELATIVE RELATION CLASS)

MAKEUNIT creates a new unit. UNIT is the name for the new unit, RELATIVE is the name of the unit that is to be the parent (generalization or prototype) of the new unit. RELATION is the proposed relation to RELATIVE, either SPEC or INST. CLASS, if given, is a group classification (or list of group classifications) for the new unit. (In addition, the new unit will inherit the group classifications of RELATIVE.) The new unit also inherits the slots that appear in RELATIVE. MAKEUNIT returns the name of the new unit if successful, otherwise NIL.

### 5.2.3 DELETEUNIT (UNIT KEEPSPECFLG MSGFLG)

DELETEUNIT deletes a unit from the knowledge base. UNIT is the name of the unit. All instances of UNIT are automatically deleted. The treatment of specializations of the deleted unit depends on the value of KEEPSPECFLG. If KEEPSPECFLG = NIL, all specializations of UNIT are deleted as well. If KEEPSPECFLG = T, DELETEUNIT will modify the knowledge base so that specializations of UNIT become specializations of its parent. However, any slots that were defined in UNIT will disappear from the specializations. If MSGFLG is T, then ***DELETE*** messages will be sent to slots as appropriate. The value of DELETEUNIT is UNIT if successful, otherwise NIL.

### 5.2.4 RENAMEUNIT (OLDNAME NEWNAME FIXFLG)

RENAMEUNIT changes the name of a unit from OLDNAME to NEWNAME. It updates all generalization hierarchy relations in the knowledge base. It returns NEWNAME as its value if the name change was successful, and NIL otherwise. RENAMEUNIT correctly renames description and indefinite units created from OLDNAME and fixes their ***DESCR*** slots. If FIXFLG = T then all references to OLDNAME (and its corresponding description and indefinite units) in the slots of other units in the knowledge base are fixed. (NOTE: This is a *slow* process. It necessitates a search through the entire knowledge base.)

### 5.2.5 INSTANTIATE (UNIT)

INSTANTIATE converts UNIT to an instance of its parent if it is currently a specialization. (This changes a definition of a class [i.e., branch of the generalization hierarchy] to a definition of an individual.) The operation can only be accomplished if UNIT has no progeny and no ***R*** slots that are not filled with terminal values. INSTANTIATE returns UNIT if successful, otherwise NIL.

### 5.2.6 SPECIALIZE (UNIT)

SPECIALIZE converts UNIT to a specialization of its parent if it is currently an instance. (This changes a definition of an individual to a definition of a class [i.e., a branch of the generalization hierarchy].) SPECIALIZE returns UNIT if successful, otherwise NIL.

### 5.2.7 COPYUNIT (FROMUNIT TOUNIT)

COPYUNIT copies all of the information in FROMUNIT to TOUNIT. If TOUNIT is not specified, a new unit name will be assigned (constructed from FROMUNIT) and the new unit will be created. The value of COPYUNIT is TOUNIT if successful (or the constructed name of the new unit if TOUNIT is NIL), otherwise NIL. (COPYUNIT correctly copies all datatypes.)

### 5.2.8 MOVEUNIT (UNIT PARENT PFLG)

MOVEUNIT moves UNIT so that PARENT becomes its parent. If PFLG is T then progeny are also moved.

If PFLG is NIL, then all instances are deleted, and specializations become specializations of the old generalization of the unit. Top-level slots in the unit are deleted in its old specializations.

If any conflicts exist between slots in UNIT and slots of PARENT, a message is displayed. Slots that are inherited from the new parent take precedence over slots with the same name in the unit to be moved (or its progeny). The offending slots in the unit are not lost. They are copied into top-level slots with names of the form "$$-<slotname>."

### 5.2.9 SPLITUNIT (OLDUNIT TOPUNIT TOPSLOTS)

SPLITUNIT is used to divide a unit into two units--thus creating a new level in the generalization hierarchy. After this operation a new unit (TOPUNIT) containing some of the slots of OLDUNIT will appear in the generalization hierarchy as the generalization of OLDUNIT.

OLDUNIT must be an existing unit in the knowledge base and be a specialization of some other unit. (It cannot be an instance.) TOPUNIT must be a valid unit name but must not already exist in the knowledge base.

TOPUNIT will acquire all of the slots that were inherited from above by OLDUNIT plus all of the slots in the list TOPSLOTS. Each slot in TOPSLOTS must be a top-level slot in OLDUNIT. At the end of this operation, each of the slots in TOPSLOTS will be a top-level slot in TOPUNIT and will be inherited by OLDUNIT.

All specializations and instances of OLDUNIT will remain unchanged except that slots in TOPSLOTS will now be inherited from TOPUNIT instead of from OLDUNIT. OLDUNIT will become a specialization of TOPUNIT and the old generalization of OLDUNIT will become the generalization of TOPUNIT.

If any of the conditions above are not met, SPLITUNIT returns NIL and makes no changes to the knowledge base. If successful, SPLITUNIT returns TOPUNIT as its value.

### 5.2.10 MAKEUNITNAME (PREFIX)

MAKEUNITNAME provides a sort of GENSYM for creating new unit names. This is useful for routines which must create new units as part of their processing. MAKEUNITNAME returns as its value a unit name starting with the given PREFIX and followed by a number. For example, (MAKEUNITNAME SAMPLE) would return SAMPLE2 if SAMPLE1 already existed as a unit in the knowledge base. MAKEUNITNAME does not create the new unit; it only supplies a name.

### 5.2.11 UNITNAMESORT (UNIT1 UNIT2)

UNITNAMESORT is used for sorting unit names. It returns T if UNIT1 should be placed ahead of UNIT2 in a sorted list. UNITNAMESORT gives a more reasonable ordering for unit names constructed by MAKEUNITNAME than normal alpha ordering (e.g. WIRE2 will appear before WIRE19).

## 5.3 Functions For Computing Built-in Relations On Units

### 5.3.1 LISTRELATIVES (UNIT RELATION CLASS)

LISTRELATIVES returns a list of the names of units having a particular RELATION to UNIT in the generalization hierarchy. RELATION may be one of the following: INST (for Instance), PROTO (for prototype), GEN (for generalization) or SPEC (for specialization).

If the optional argument CLASS is given, only those units having that group classification will be included.

IF RELATION is SPEC or INST, then a *copy* of the actual list of relatives is returned by LISTRELATIVES.

### 5.3.2 GEN (UNIT)

GEN returns the name of the generalization of UNIT. If UNIT doesn't exist or has no generalization (e.g., is an instance), GEN returns NIL.

### 5.3.3 PROTO (UNIT)

PROTO returns the prototype of UNIT. If UNIT doesn't exist or has no prototype (e.g., is a specialization), PROTO returns NIL.

### 5.3.4 PARENT (UNIT)

PARENT returns the prototype of UNIT if it is an instance or the generalization of UNIT if it is a specialization.

### 5.3.5 SPEC (UNIT CLASS)

SPEC returns a *copy* of the list of the immediate specializations of UNIT. If CLASS is specified, only those specializations having that group classification are included.

### 5.3.6 INST (UNIT CLASS)

INST returns a *copy* of the list of all instances of UNIT. If CLASS is specified, only those instances having that group classification are included.

### 5.3.7 SPEC* (UNIT CLASS)

SPEC* returns a list of all specializations of UNIT and all specializations of them (recursively). In other words, it returns a list of all progeny of UNIT that are not instances. If the optional argument CLASS is given, only those units having that group classification are included.

### 5.3.8 INST* (UNIT CLASS)

INST* returns a list of all instances of UNIT and all instances of its specializations. If the optional argument CLASS is given, only those units with that group classification are included.

### 5.3.9 PROGENY* (UNIT CLASS)

PROGENY* returns a list of all specializations and instances of UNIT (to all depths). If the optional argument CLASS is given, only those units having that group classification are included.

### 5.3.10 PROGENY? (UNIT)

PROGENY? returns T if UNIT has progeny. It is fast and generates no list structure.

### 5.3.11 PROGENYCLASS? (UNIT CLASS)

PROGENYCLASS? returns T if UNIT has any progeny with group classification CLASS. It generates no list structure.

### 5.3.12 ANCESTOR? (UNIT ANCESTOR)

ANCESTOR? returns ANCESTOR if ANCESTOR is a (possibly distant) generalization of UNIT, otherwise NIL. For convenience, if UNIT is an instance, ANCESTOR? performs the test for its prototype.

### 5.3.13 INST*? (UNIT INST)

INST*? returns T if INST is a (possibly distant) instance of UNIT (i.e., an instance of UNIT or or an instance of a [perhaps distant] specialization of UNIT), otherwise NIL.

### 5.3.14 GENLEVEL (UNIT ANCESTOR)

GENLEVEL counts the number of generations between UNIT and its ANCESTOR. If ANCESTOR is not given, the ROOT is assumed. This function may be used as a measure of how *specialized* a unit is. If any error is detected, NIL is returned.

## 5.4 Functions That Pertain To Group Classification Of Units

### 5.4.1 CLASS? (UNIT CLASS)

CLASS? is used for inquiries about the group classification of a unit. If UNIT is the only argument, then CLASS? returns as its value a list of all of the groups to which UNIT belongs. If the CLASS argument is also specified, and is an atom, then CLASS? acts as a boolean test. If CLASS is a list, then CLASS? will return T if any atom in the list is a group to which UNIT belongs.

### 5.4.2 CLASSIFY (UNIT CLASS ONLYFLG)

CLASSIFY adds CLASS (which may be a list) to the group classification list for UNIT (unless UNIT already has that classification). Unless ONLYFLG = T, CLASSIFY propagates group classifications to all progeny of a unit. The value of CLASSIFY is UNIT if successful and NIL otherwise.

### 5.4.3 DECLASSIFY (UNIT CLASS ONLYFLG)

DECLASSIFY is the inverse of CLASSIFY. It takes the same arguments. DECLASSIFY returns UNIT if successful and NIL otherwise.

### 5.4.4 COPYGROUP (FROMGROUP TOGROUP)

COPYGROUP is useful for copying groups of units. For each unit that is a member of FROMGROUP (i.e., has FROMGROUP in its list of group classifications), COPYGROUP creates a unit (using COPYUNIT [Section 5.2.7]) with group classification TOGROUP. Any pointers between units in FROMGROUP are replaced by corresponding pointers to units in TOGROUP.

## 5.5 Functions That Operate On Slots

### 5.5.1 SLOT? (SLOT UNIT)

SLOT? returns SLOT if it is the name of a slot in UNIT, and NIL otherwise.

### 5.5.2 MAKESLOT (SLOT UNIT ROLE DATATYPE)

MAKESLOT creates a new slot with name SLOT in UNIT and all of its progeny. ROLE is the inheritance role for the slot and DATATYPE is its datatype. (ROLE and DATATYPE is optional.) Since slots that are defined in a unit (termed top-level slots) using MAKESLOT will appear in all of its progeny, MAKESLOT first checks whether the slot is used in any of the progeny of UNIT. (PROGENYSLOT? is used to check this [see Section 5.5.7].) If the slot appears in progeny and there is a conflict in either ROLE or DATATYPE, then MAKESLOT reports an error. MAKESLOT returns SLOT if successful, otherwise NIL.

NOTE: MAKEUNIT (Section 5.2.2) automatically creates appropriate slots (inherited from ancestors) at the time a unit is created. Thus it is not necessary for the user to call MAKESLOT to create those slots that are already defined in the parent of a new unit.

### 5.5.3 DELETESLOT (SLOT UNIT KEEPFLG MSGFLG)

DELETESLOT deletes SLOT from UNIT and its progeny. This operation is valid only for top-level slots. The extent of the deletion is controlled by KEEPFLG as follows:

NIL (Default case): Delete SLOT in all progeny.

T: Delete SLOT in UNIT but keep it in all progeny.

NOTSPEC: Delete SLOT in UNIT and its immediate instances. Keep SLOT in specializations and their progeny.

If MSGFLG is T, then ***DELETE*** messages will be sent to the slot as appropriate.

DELETESLOT returns SLOT if successful and NIL otherwise.

### 5.5.4 RENAMESLOT (SLOT UNIT NEWNAME)

RENAMESLOT renames SLOT as NEWNAME in UNIT and its progeny. This operation is valid only for top-level slots. RENAMESLOT detects an error if a NEWNAME slot already exists in UNIT or in its progeny. RENAMESLOT returns NEWNAME if successful and NIL otherwise.

### 5.5.5 SORTSLOTS (UNIT SLOTLIST SORTFLG)

No meaning is attached by the UNIT Package to the order in which slots appear in a unit. It is sometimes convenient, however, for users to have the slots appear in a particular order. SORTSLOTS changes the internal ordering of slots in UNIT according to SLOTLIST and SORTFLG. (This changes the order in which they will be presented by LISTSLOTS [Section 5.5.6].)

The following methods for sorting the slots are provided:

| SLOTLIST | SORTFLG | Effect |
|---|---|---|
| NIL | NIL | Alphabetical Order of Slots. |
| Given | Ignored | Order given in SLOTLIST. |
| NIL | H | Hierarchical ordering. (See below.) |
| NIL | R | Reverse Hierarchical ordering. |

Hierarchical ordering means an ordering of slots depending on the level in the unit hierarchy from which they have been inherited. Slots are ordered so that those which are inherited from the highest place in the generalization hierarchy appear first. Slots inherited from the same unit are placed in alphabetical order.

SORTSLOTS returns UNIT if successful and NIL otherwise.

### 5.5.6 LISTSLOTS (UNIT FIELD VALUE)

LISTSLOTS returns a list of the names of all of the slots in UNIT having a FIELD with the specified field VALUE. If FIELD = NIL, then LISTSLOTS simply returns a list of the names of all of the slots in the unit.

A special case is recognized in the case that FIELD = ROLE, which makes this function somewhat more useful. When FIELD = ROLE and VALUE is a single role, LISTSLOTS returns all slots for which (ROLE? SLOT UNIT VALUE) = T; that is, all slots having that inheritance role. (The point is that the slots may have other roles too--so that the criterion is membership and not exact equality.) When VALUE is a list of roles, an analogous criterion is used.

### 5.5.7 PROGENYSLOT? (SLOT UNIT FIRSTFLG)

PROGENYSLOT? returns a list of progeny of UNIT that have a slot with name SLOT. Search is limited according to FIRSTFLG as follows:

NIL (Default case): List of all progeny having the slot.

T: Stop search as soon as *any* progeny is found having the slot.

TOPLEVEL: Return a list of all progeny in which the slot is a top-level slot; that is, those progeny in which the slot is first defined.

### 5.5.8 TOPLEVELUNIT? (SLOT UNIT)

TOPLEVELUNIT? returns the name of the ancestor in which SLOT in UNIT was defined. In case of error (e.g., slot doesn't exist), TOPLEVELUNIT? returns NIL.

### 5.5.9 TOPLEVELSLOT? (SLOT UNIT)

TOPLEVELSLOT? returns T if SLOT is first defined in UNIT and NIL otherwise. In other words, if the slot is inherited from a parent, TOPLEVELSLOT? returns NIL.

Both TOPLEVELUNIT? and TOPLEVELSLOT? can be used to determine whether a given slot is first defined in a given unit. However, TOPLEVELSLOT? is more efficient for this purpose because it only looks at the parent of the given unit instead of tracing back through an arbitrary number of ancestors.

### 5.5.10 INHERITEDSLOT? (SLOT UNIT)

If SLOT in UNIT is *directly* inherited (i.e., its inheritance role is ***S***) from either a prototype or a generalization, INHERITEDSLOT? returns the name of the (possibly distant) ancestor in which SLOT is defined. For the case of dual role slots--(***S R***) or (***S O***)--INHERITEDSLOT? returns the name of the ancestor in which the ***S*** role was added. Otherwise it returns NIL. (Inherited slots cannot be modified unless changes are made in that ancestor.)

NOTE: INHERITEDSLOT? will return NIL for a top-level slot whose role is ***S***--on the basis such a slot is not itself inherited. However, for each of the progeny of unit in which the slot is defined, INHERITEDSLOT? will return the name of the top-level unit.

### 5.5.11 UNCHANGEDSLOT? (SLOT UNIT)

If SLOT in UNIT is in ***NC*** format (see Section 6.3.3), UNCHANGEDSLOT? returns the name of the master unit that is being tracked for the slot. If SLOT is not in ***NC*** format, UNCHANGEDSLOT? returns NIL.

### 5.5.12 TRACKINGSLOT? (SLOT UNIT)

If SLOT in UNIT is *tracking* another unit, then TRACKINGSLOT? returns the name of that unit, otherwise NIL. (This is an effective OR of INHERITEDSLOT? and UNCHANGEDSLOT?.)

### 5.5.13 SLOT-MENTIONS (SLOT UNIT CLASS SKPLST REFSFLG)

SLOT-MENTIONS returns a list of all of the units mentioned in SLOT of UNIT. If CLASS is specified, only those units having that group classification are included. Units that are members of the ***PRIMITIVE*** group (i.e., datatypes) are not included. Units on SKPLST are

ignored. If REFSFLG is NIL, then references to ***DESCR*** and ***INDEFINITE*** units are traced to their parent units. SLOT-MENTIONS checks the value and default fields of SLOT. (The optional fields cannot be checked properly since they have no datatype. Hence atoms cannot be distinguished from units.) SLOT-MENTIONS uses the UNITS procedure attached to the datatype unit for SLOT to parse the fields.

### 5.5.14 UNIT-MENTIONS (UNIT CLASS REFSFLG)

UNIT-MENTIONS returns a list of all units mentioned in the slots of UNIT. If CLASS is specified, only those units having that group classification are included. If REFSFLG is NIL, then references to ***DESCR*** and ***INDEFINITE*** units are traced to their parent units.

### 5.5.15 SHOWREFS (UNIT CLASS REFSFLG)

SHOWREFS returns a list of all units whose slots refer to UNIT. If CLASS is given, then only units with that group classification are searched. If REFSFLG is NIL, then references to ***DESCR*** and ***INDEFINITE*** units are traced to their parent units.

### 5.5.16 FIX-SLOTREFS (UNIT OLDNAME NEWNAME)

FIX-SLOTREFS changes all references to OLDNAME in the slots of UNIT to be references to NEWNAME.

## 5.6 Functions That Operate On Fields

### 5.6.1 FIELD? (FIELD SLOT UNIT)

FIELD? returns FIELD if it is a field of SLOT in UNIT. Otherwise it returns NIL.

### 5.6.2 LISTFIELDS (SLOT UNIT NOT-FIXED-FIELDS-FLG)

LISTFIELDS returns a list of the fields for SLOT in UNIT. If NOT-FIXED-FIELDS-FLG = T, then only the optional fields are included.

### 5.6.3 MAKEFIELD (FIELD SLOT UNIT)

MAKEFIELD creates FIELD in SLOT of UNIT. An error results if FIELD corresponds to one of the permanent fields or is already a field of SLOT in UNIT. (Since PUTFIELD (Section 5.6.6) will create a field if it is not there already, this function is not strictly necessary and is included only for completeness.)

### 5.6.4 DELETEFIELD (FIELD SLOT UNIT)

DELETEFIELD removes FIELD from SLOT of UNIT. An error occurs if this is attempted for a permanent field.

### 5.6.5 GETFIELD (FIELD SLOT UNIT)

GETFIELD returns the value stored in FIELD of SLOT in UNIT. It works for both permanent and optional fields. GETFIELD does not copy values as they are returned from fields of a slot. For most cases this is no concern. However, when these values are list structures, the user should *copy* them before altering them. Failure to do so can result in bizarre effects--such as simultaneous changes in the ancestor units that contain the slot.

### 5.6.6 PUTFIELD (VALUE FIELD SLOT UNIT PFLG)

PUTFIELD puts VALUE in FIELD of SLOT in UNIT. It works for permanent and optional fields except for the NAME and ROLE fields. Special checking is performed for the datatype field. PUTFIELD returns FIELD if successful and NIL otherwise.

PFLG is an optional argument that controls forwarding of the field value to progeny. If no PUTFIELD operation on progeny has been performed, they always inherit all of the fields from the corresponding slot in an ancestor (i.e., the slot in the ancestor is tracked). Once they have been changed, then value, default, and optional fields do not get forwarded unless PFLG = T.

### 5.6.7 GETROLE (SLOT UNIT)

GETROLE returns the inheritance role list for SLOT in UNIT.

### 5.6.8 ROLE? (SLOT UNIT ROLE)

Without the optional ROLE argument, ROLE? is the same as GETROLE. If ROLE is given, ROLE? returns T if ROLE is contained in the inheritance role list for SLOT in UNIT.

### 5.6.9 PUTROLE (ROLE SLOT UNIT)

PUTROLE adds ROLE to the inheritance role list for SLOT in UNIT and its progeny if it is not there already. An error occurs if an attempt is made to set two mutually exclusive roles for SLOT or if SLOT is not a top-level slot in UNIT. When the roles ***R*** or ***O*** are set, the corresponding slots in progeny are initially marked as *unchanged* (***NC*** format--see Section 6.3) so that their fields will track the fields in this unit. PUTROLE returns ROLE if successful and NIL otherwise. An ***S*** role cannot be added with PUTROLE unless it is done at the same time as an ***R*** or ***O*** role. The CHANGEROLE function handles this case.

### 5.6.10 CHANGEROLE (ROLE SLOT UNIT)

CHANGEROLE changes the existing inheritance role list of SLOT in UNIT. CHANGEROLE returns ROLE if successful and NIL otherwise.

If ROLE is identical to the existing role, no operation takes place.

If ROLE = ***S*** and the existing role is ***R*** or ***O***, then the ***S*** role is added to UNIT and all of its progeny. The corresponding slots in the progeny are changed to track the SLOT in UNIT. Once the ***S*** role has been added, INHERITEDSLOT? (Section 5.5.10) will return UNIT for SLOT in the progeny. Any attempt to independently change the value fields of corresponding slots in the progeny will result in an error unless DELETEROLE (see below) is first used to delete the ***S*** role from SLOT in UNIT. This CHANGEROLE operation is valid on a slot at any level.

If ROLE = ***O*** and the existing role is ***R*** (or vice versa), then the role change is made in UNIT and in all progeny. All corresponding slots in the progeny that are marked as *unchanged* or *inherited* retain these markings (see Section 6.3). This is valid only for a top-level slot.

If ROLE = ***S*** and the existing role is ***U***, then all corresponding slots in the progeny are made to track SLOT in unit. They become *inherited* slots (see Section 6.3). This is only valid for a top-level slot.

Other Combinations: If ROLE = ***R*** or ***O*** and the existing role is ***U*** or ***S***, or if ROLE = ***U*** and the existing role is ***S***, ***R***, or ***O***, then the role is changed in UNIT and in all progeny (and slots are remade to be *tracking* slots or *standard* slots as required by the case (see Section 6.3). This is only valid for a top-level slot.

**NOTE**: This is a very primitive function for changing roles and does not take into account any of the changes that may be necessary for consistency of restrictions or use of terminal values. The functions EDITSLOT, EDITBADVALUES and CLEARBADVALUES (see Section 5.7) provide more assistance when major knowledge base changes are being performed.

### 5.6.11 DELETEROLE (ROLE SLOT UNIT)

DELETEROLE deletes ROLE from the inheritance role list for SLOT in UNIT if it is there. Except in the special case where the role being deleted is ***S*** from a slot having ***S*** and another role, an error message is generated if the slot is not a top-level slot. Deleting a role will often have very widespread consequences for the progeny of the current unit as explained in the following table. DELETEROLE returns ROLE if successful and NIL otherwise.

***S*** (Special ***S*** and ***R*** or ***S*** and ***O*** case):

The ***S*** role is deleted from SLOT in UNIT. All of the corresponding slots in the progeny of UNIT are marked as *unchanged* (see Section 6.3).

***S***, ***R***, ***O***, ***U***:

ROLE is deleted from SLOT in UNIT and all of its progeny. The value, default, and optional fields are set to NIL in UNIT and its progeny. If some instance of SLOT in one of the

progeny has a dual role (*S* and *R* or *S* and *O*), the *S* role is deleted along with the other role. At the conclusion of this operation, none of the instances of SLOT in the progeny will be *tracking* SLOT in UNIT.

**NOTE:** The function DELETESLOT (Section 5.5.3) will remove a slot from a unit and all of its progeny. Hence, the last case above is expected to have only limited utility.

### 5.6.12 DATATYPE? (SLOT UNIT DATATYPE)

DATATYPE? returns the datatype of SLOT in UNIT. If DATATYPE is not given, this is equivalent to the appropriate call to GETFIELD. If DATATYPE is given, then DATATYPE? returns T if the datatype of SLOT is DATATYPE and returns NIL otherwise.

## 5.7 Functions For Operating On Values

### 5.7.1 GETVALUE (SLOT UNIT)

GETVALUE returns the value for SLOT in UNIT. When SLOT has a simple stored value, it is simply returned. GETVALUE has been extended to allow for virtual slots and computed values as follows:

| Case | Action |
|---|---|
| Simple Stored Value | Return the value. |
| SLOT exists, has no value, but has a *TOGET* field | Invoke the procedure in the TO-GET field and return its value. (Note that if SLOT has a value, this procedure will not be called.) If the value of the TO-GET field is an atom, it is applied as follows: (APPLY* VAL UNIT). If it is a list, it is assumed to be of the form (FN ARG1 ARG2 ...), and is applied as follows: (APPLY (CAR VAL) (APPEND (CDR VAL) (LIST UNIT))). |
| SLOT doesn't exist, but there is a procedure named SLOT (i.e., a procedure with the same name as the slot). | Invoke the procedure and return the value: (APPLY* SLOT UNIT). |
| SLOT doesn't exist, but there is a unit with the same name as the slot | Invoke GETVALUE recursively on slot named VALUE in that unit. This could result in a constant or a computed value being returned. |

GETVALUE does not copy values as they are returned from a slot. For most cases this is of no concern. However, when these values are list structures, the user should *copy* them before altering them. Failure to do so can result in bizarre effects--such as simultaneous changes in the ancestor units that contain the slot.

### 5.7.2 PUTVALUE (SLOT UNIT VALUE NOTESTFLG ROLE TV PFLG)

PUTVALUE puts VALUE in the value field of SLOT in UNIT. Unless NOTESTFLG = T, PUTVALUE verifies that the inheritance role and terminal value requirements are satisfied before putting VALUE in the value field of the SLOT in UNIT.

If ROLE is given it is used as the inheritance role. If TV = T, then VALUE is assumed to be terminal, else it is checked. If PFLG = T then the value is propagated to the progeny of the unit.

If VALUE is a terminal value and the role is either ***R*** or ***O***, PUTVALUE adds the role ***S*** using CHANGEROLE (Section 5.6.10) if the role ***S*** is not set already. Conversely, if VALUE is not terminal and the role is ***S*** and either ***R*** or ***O***, PUTVALUE will delete the ***S*** role. This action assures that slots having terminal values will consistently also have the role ***S***.

PUTVALUE returns SLOT if successful and NIL otherwise. If there is an error, the value in the slot is not changed.

### 5.7.3 GETVORD (SLOT UNIT FASTFLG)

GETVORD typically returns the default value for SLOT in UNIT if the value field is not filled. More precisely, its operation depends on FASTFLG as follows:

FASTFLG = NIL (usual case): If the value of SLOT is terminal (not a value-restriction), then that value is returned. Otherwise the default value of SLOT is returned.

FASTFLG = T: GETVORD returns (OR VALUE DEFAULT); that is, it does not test whether the value of SLOT is terminal.

### 5.7.4 EDITSLOT (SLOT UNIT FAKERESTRICTION FAKEVALUE NOEFLG)

EDITSLOT causes the value of SLOT in UNIT to be edited interactively, using the editor associated with the datatype of the slot. If the optional argument FAKERESTRICTION is given, it will be used in place of the actual restriction on the slot. If the optional argument FAKEVALUE is given, then it will be passed to the datatype editor in place of the value in the slot. If NOEFLG is T, then the datatype editor is not actually called. Instead, the existing value is checked for consistency as a value for SLOT in UNIT. (This is used by the ***CONSISTENCY*** command [Section 3.3.2].)

EDITSLOT first invokes the editor associated with the datatype to acquire the value in

the slot or permit the user to modify the existing value. After the local editor has returned a new value, EDITSLOT checks the new value for possible conflicts with restrictions inherited from above in the generalization hierarchy. (This extra checking is not done if the editor returns the value NOTEST.)

After checking for conflicts from above, EDITSLOT then checks for conflicts below. If the value in the slot conflicts with the corresponding values in progeny, EDITSLOT again interacts with the user. He may display the conflicts, re-edit the slot, re-edit the progeny in which there are conflicts, or clear all of the slots in progeny in which there is a conflict. EDITSLOT also verifies that the values left in instances are terminal values.

### 5.7.5 PRINTSLOT (SLOT UNIT FAKEVALUE NOCRFLG POSLST)

PRINTSLOT causes the value of SLOT in UNIT to be printed using the printing routine associated with the datatype of the slot. If FAKEVALUE is given, it is passed along to the datatype printing routine instead of the actual value in the slot. If NOCRFLG is given, it is also passed along to the datatype printing procedure. By convention, the datatype printing routines terminate printing with a carriage return unless NOCRFLG = T. POSLST, if given, is also passed to the datatype printing procedure. By convention, these routines expect POSLST to be a list of fallback character positions. [When a datatype printing procedure is called, and the current position is too large to allow printing any integral part of the instance of the datatype on the current line without overflow, then the datatype printing procedure tries to fall back to successive character positions found on POSLST (in order of decreasing magnitude) to find a position that does allow non-overflow printing.]

### 5.7.6 GETRESTRICTION (SLOT UNIT ROLE)

GETRESTRICTION returns the restrictions on the value of the SLOT in UNIT. If ROLE is given, it is used in place of the inheritance role of SLOT. Restrictions are located depending on the inheritance role as follows: If the role is *U* or SLOT is a top-level slot, then NIL is returned. Otherwise, the value of SLOT in the parent of UNIT is returned.

### 5.7.7 TERMINALVALUE? (SLOT UNIT FAKEVALUE)

TERMINALVALUE returns T if the value of SLOT in UNIT is a terminal value, NIL if it is a value restriction. If FAKEVALUE is given, the test is performed as if FAKEVALUE were the value in the slot. TERMINALVALUE invokes the terminal value routine associated with the datatype of the slot to perform the test.

### 5.7.8 CHECKRESTRICTION (SLOT UNIT VALUE RESTRICTION ROLE)

CHECKRESTRICTION returns NIL if VALUE satisfies the restrictions on the value field of SLOT in UNIT and the locally generated restriction error message otherwise.

The arguments VALUE, RESTRICTION and ROLE are optional. If VALUE is given, then CHECKRESTRICTION returns the message that would occur if VALUE were the value in SLOT. Similarly, if RESTRICTION is given, it is used in place of the restrictions returned by GETRESTRICTION. Finally, if ROLE is given, then it is used in place of the actual inheritance role of SLOT. CHECKRESTRICTION immediately returns NIL for ***S*** and ***U*** roles.

CHECKRESTRICTION uses SLOTMSG (Section 6.8.2) to check for conflicts if the restriction is not NIL by sending a ***STRICTER*** message that activates the attached procedures that specialize in the restrictions for the particular datatype.

NOTE: To help distinguish between the case where the value satisfies the restrictions (NIL returned) and an error occurs during the operation of CHECKRESTRICTION (NIL returned), CHECKRESTRICTION first sets the UNIT Package global error number, ***UA.ERRNO***, to NIL. ***UA.ERRNO*** will still be NIL only after an error-free operation.

### 5.7.9 PROGENYRESTRICTION (SLOT UNIT RESTRICTION FIRSTFLG ROLE)

PROGENYRESTRICTION tests for conflicts below. In other words, PROGENYRESTRICTION tests whether a value restriction in SLOT of UNIT conflicts with values in the corresponding slots in the progeny of UNIT.

If RESTRICTION is given, then it is used rather than the contents of the value field of SLOT in UNIT. Similarly, if ROLE is given, then it is used in place of the inheritance role of SLOT in UNIT.

PROGENYRESTRICTION returns as its value a list of the next generation of progeny for which RESTRICTION would cause a conflict. If FIRSTFLG = T, PROGENYRESTRICTION stops its search for conflicts as soon as it has found the first conflict. If there are no conflicts, PROGENYRESTRICTION returns NIL.

Like CHECKRESTRICTION (and for the same reason), PROGENYRESTRICTION first sets ***UA.ERRNO*** to NIL.

### 5.7.10 CLEARBADVALUES (SLOT UNIT RESTRICTION)

CLEARBADVALUES causes all slots corresponding to SLOT in the progeny of UNIT to be examined to see whether they conflict with the value in SLOT in UNIT. If the optional argument RESTRICTION is given, it is used in place of the actual value of SLOT in UNIT. After the corresponding SLOT in progeny of UNIT are checked, values that do not satisfy the restriction are set to NIL. CLEARBADVALUES returns SLOT if its operation is successful and NIL otherwise.

### 5.7.11 EDITBADVALUES (SLOT UNIT RESTRICTION)

EDITBADVALUES causes an appropriate editor to be invoked on all slots corresponding to SLOT in the progeny of UNIT that conflict with the value of SLOT in UNIT. If the optional argument RESTRICTION is given, it is used in place of the actual current restriction in SLOT. EDITBADVALUES returns SLOT if the operation is successful and NIL otherwise.

## 5.8 Message Functions

### 5.8.1 UNITMSG (UNIT TOKEN ARG1 ... ARGN)

UNITMSG sends a message to UNIT. The message handler is assumed to be the value of the slot named TOKEN in UNIT. If the value is not a procedure, then the value is simply returned. If the value is a procedure, then UNITMSG applies the procedure with the arguments ARG1 ... ARGN.

UNITMSG returns as its value the value of the procedure activated as described. In case of error, UNITMSG generates a diagnostic message and returns NIL. It is usually useful for attached procedures to return a non-NIL value after a successful execution. However, to help distinguish a NIL value returned from an error condition UNITMSG also first sets *UA.ERRNO* to NIL.

### 5.8.2 SLOTMSG (SLOT UNIT TOKEN FAKEVALUE ARG2 ARGLIST)

SLOTMSG is responsible for sending a message to a particular slot. SLOTMSG looks for the procedure (or value) indexed under TOKEN as follows:

1. In a field named TOKEN of SLOT in UNIT. (Must be an optional field).

2. In the slot named TOKEN of the datatype unit corresponding to SLOT of UNIT.

3. In the slot named TOKEN of the unit named SLOT.

In performing this search, if a value is found that is not a procedure, SLOTMSG simply returns that value. Otherwise, the procedure is applied to the arguments described below:

Arg1: VALUE
Arg2: ARG2
Arg3: SLOT
Arg4: UNIT
Arg5-N: Elements of the list ARGLIST.

It is sometimes useful to send a message as if the value in the slot were some hypothetical value. For example, if one might want to check the effects of an attached procedure for checking restrictions on some proposed new value for the slot. To do this, the

FAKEVALUE argument may be used. If FAKEVALUE is given, SLOTMSG will deliver its message with the first argument set to FAKEVALUE instead of the actual value in the slot.

Like UNITMSG, SLOTMSG returns as its value the value of the procedure activated as described. In case of error, SLOTMSG generates a diagnostic message and returns NIL. Because of this, it is usually helpful if attached procedures return some non-NIL value after a successful execution. However, to help distinguish a NIL value returned from an error, SLOTMSG also first sets ***UA.ERRNO*** to NIL.

### 5.8.3 MSGHANDLER? (SLOT UNIT TOKEN)

The message handling mechanism described above typically involves a search process to locate the procedure that should be activated by a given message. In most cases the procedure is located in the datatype unit for the slot but it can also be found in a field associated with the slot. MSGHANDLER? carries out this SLOTMSG search process for SLOT of UNIT and returns as its value the name of the unit that has the procedure. MSGHANDLER? returns NIL if no procedure is found. MSGHANDLER? can thus be used to determine whether SLOT in UNIT can respond to a TOKEN message.

## 5.9 Functions That Operate On Datatypes

These are some basic retrieval functions for the ***UNIT*** and ***LIST*** datatypes. Some of them operate on slots, some operate on units, and others operate on descriptions. Some search the knowledge base for units which match a description and others just return the descriptions. (The term *description* is used in a general way in this section. It refers to all of the types of value that can be set with the Unit datatype editor [yonsss uedit)], and *not* just to description units.)

### 5.9.1 GETMATCHES (SLOT UNIT)

GETMATCHES returns a list of units that match the description in SLOT of UNIT. It computes progeny or instances, follows references, and performs matches as needed. The value of GETMATCHES is always a list even if there is only one unit referenced. This is the most general retrieval function for the ***UNIT*** datatype; it makes use of the more specialized functions below. It provides for retrieval of the list of units referenced by a slot in a manner which is independent of the form which is stored.

### 5.9.2 GETUNITS (SLOT UNIT)

GETUNITS is similar to GETMATCHES but differs in that it returns the descriptions (simplified) instead of the units that match the descriptions. More precisely, it returns a list of descriptions as generated by GETUNIT.

### 5.9.3 GETUNIT (DESCR)

GETUNIT is useful for returning a description without performing matches. It returns a unit as follows:

| Form of Description | What GETUNITS returns |
|---|---|
| Unitname | Unitname |
| List of units | First unit in the list. (Oddball case) |
| *P Unitname | Unitname |
| *I Unitname | Unitname |
| *D Unitname | Unitname |
| *REFQ Reference | Chases the reference and recurs. |
| *M Reference | Chases the reference and recurs. |

### 5.9.4 UNITMATCH (DESCR START CLASS)

UNITMATCH returns a list of units that match DESCR (a description unit). START is optional and specifies a unit in the generalization hierarchy at which the search should begin. Units that are progeny of START are checked. If START is NIL, then the closest non-description unit parent of DESCR is assumed. If the optional argument CLASS is given, then only units having that group classification are considered.

### 5.9.5 SLOTMATCH (GEN SLOT VAL CLASS)

SLOTMATCH returns a list of all units with a slot named SLOT whose value matches VAL. The search is limited to units that are progeny of GEN. If CLASS is given, then the search is restricted to units having that group classification.

### 5.9.6 REF? (SLOT UNIT)

REF? returns *REFQ for pathname indirect references and *M for indirect references that mention an ancestor. It returns NIL otherwise.

### 5.9.7 GET-REF (REFEXPR)

GET-REF returns the value of REFEXPR, which is an indirect-reference expression. REFEXPR should be a list in one of the legal formats:

Pathname format: (*REFQ SLOT UNIT PFLG CLASS)
Ancestor format: (*M ANCESTOR UNIT CLASS)

### 5.9.8 RESOLVE-GROUP (CLASS START)

RESOLVE-GROUP resolves all references in the slots in all units with group classification CLASS. If START is specified, only those units that are progeny of START are considered. If START is NIL, ***ROOT*** is assumed. Slots of role ***U*** and directly inherited ***S*** slots are ignored. RESOLVE-GROUP uses GETMATCHES to resolve the references and REF? to select the slots.

### 5.9.9 RESOLVE-EXPR (EXPR)

RESOLVE-EXPR replaces all indirect reference expressions in EXPR with the corresponding lists of units computed by GETMATCHES.

## 5.10 LIST Datatype Functions

### 5.10.1 GET-LIST (SLOT UNIT)

GET-LIST returns the list stored in SLOT of UNIT. The list returned is an **INTERLISP**-style list. (The first two items, which are used by the **UNIT Package**, are not returned.) GET-LIST does not copy the list.

### 5.10.2 PUT-LIST (SLOT UNIT LIST)

PUT-LIST replaces the list in SLOT of UNIT with LIST. (The two **UNIT Package** tokens at the beginning are left the same.)

### 5.10.3 ADD-LIST (SLOT UNIT ITEM)

ADD-LIST adds ITEM to the end of the list stored in the value field of SLOT in UNIT if it is not already in the list. (Copies the list if necessary.)

### 5.10.4 CLEAN-LIST (SLOT UNIT)

CLEAN-LIST removes all tokens from the CDDR of the list stored in the value field of SLOT in UNIT that are not terminal values. (This is useful in cases where a list initially contains descriptions of units. During processing, ADD-LIST may be used to add actual unit names. At the end, CLEAN-LIST can be used to remove the descriptions.)

### 5.10.5 GET-LIST-DATATYPE (LST)

GET-LIST-DATATYPE returns the datatype of LST, an entity of datatype ***LIST***.

### 5.10.6 GET-LIST-ELEMENT (LST ELEMNUM)

GET-LIST-ELEMENT returns the ELEMNUM$^{th}$ item from LST, an entity of datatype ***LIST***.

# Chapter 6
# Implementation Notes

## 6.1 Units: Internal Representation

Each unit is an INTERLISP atom with the following information stored on its property list.

**USLOTS**: A list of SLOT records (described below).

**UMODIFYFLAG**: A flag that is T if the unit has been altered since it was last written to disk, else NIL.

**UTIMESTAMP**: A number that indicates the *time* at which the slots of the unit were last accessed, relative to those of other units; NIL if the unit is not memory-resident.

**UDISKADDR**: The disk address of the slots of the unit (on the UNITS file).

**UGEN**: The name of the generalization of the unit if a specialization, else NIL.

**UPROTO**: The name of the prototype of the unit if an instance, else NIL.

**USPEC**: A list of the names of the specializations of the unit.

**UINST**: A list of the names of the instances of the unit.

**UCLASS**: A list of groups of which the unit is a member.

These properties are assigned and removed so as not to disturb any other properties associated with the atom. This greatly reduces the possibility of conflicts due to a unit and an atom with no relation to the UNIT Package having the same name. (This is also the reason for the "U" in front of each property name.)

The slot information is paged out to disk when space gets low. (See Section 6.2.) The remaining properties are permanently memory-resident. This implementation offers a reasonable tradeoff of space and time efficiency.[1]

Each slot is a SLOT record with the following fields:

**NAME**: The name of the slot.

---

[1] Group membership, for example, could have been indicated by a *member-of* slot in each member, or a *members* slot in each unit that stands for the group. Our implementation makes it unnecessary to search membership lists when units are deleted and makes it possible to determine whether a unit is in a group without requiring the slots of the unit to be memory-resident.

**VALUE**: The value of the slot.

**ROLE**: The inheritance role of the slot.

**DATATYPE**: The datatype of the value of the slot.

**DEFAULT**: The default value of the slot.

**FIELDS**: An association list of optional (user-defined) fields.

It is assumed that the number of slots associated with a unit is small (usually < 20). It is therefore reasonably efficient to store the slots as a list of records on the property list of a unit and locate them via linear search.

## 6.2 Unit Memory Management and Disk Representation

The **UNIT Package** contains a mechanism to use a disk file as an extended memory. Slot information is paged in and out of memory on a demand basis. This allows the use of a knowledge base that is considerably larger than the available space in memory. The basic mechanism can be augmented in packages that use the **UNIT Package** with other mechanisms to summarize, abstract, and delete data that is no longer directly related to an ongoing analysis.

The specialization, generalization, prototype, and instance relations are always memory resident as are the group classifications for a unit and information about its disk address, relative time of slot access, and whether the slots have been modified since they were last written to disk. The slots of a unit are loaded only when they are accessed.

During an INTERLISP garbage collection, if not more than ***UA.GCTRPCOUNT*** list cells are free then a flag is set (***UA.BUMPFLG***). The next time an access is made to the slots of a unit, the function UA-BUMPUNITS (Section 8.1.2) is called to move unit slots to disk (***UA.BUMPN*** at a time) until there are again at least ***UA.GCTRPCOUNT*** free cells or there are no units left whose slots can be paged out.

There are any number of criteria that could be used to determine the units whose slots are to be paged out. UA-BUMPUNITS calls a function whose name is the value of ***UA.BUMPFN***. This function is passed two arguments: ***UA.INCORELIST*** (the list of units whose slots are in memory), and ***UA.UNITLIST*** (the list of all units in the knowledge base). The function is expected to return a list of units, sorted so that the slots of those at the head of the list are to be paged out before the slots of those at the end of the list. The default function is UA-TIMESORT, which sorts the units according to recency of access (Section 8.1.43).

When the slots of a unit are paged out to disk, they are stored on the UNITS file. This file is organized as a collection of physically adjacent blocks. The blocks are either *reserved* (i.e., contain unit slots), or *free* (i.e., available for use). Each block has a header record and a trailer record with the unit slots in between (for reserved blocks). Free blocks on the disk are allocated using Knuth's boundary tag dynamic allocation scheme **[Knuth, 1968]**. Thus the free blocks are located on a doubly linked disk list.

The format of a free block is as follows:

Header Record:
TAG = F
SIZE = N (Number of characters in the block)
FLINK = Disk Address of Previous Block In Free List
BLINK = Disk Address of Next Block In Free List
Undefined Structure:
(whatever was in the block before it was declared free)
Trailer Record:
TAG = F
SIZE = N
DUMMY = nZZZ

FLINK and BLINK are pointers to the neighboring blocks on the free list. The notation "nZZZ" indicates that a number of "Z" characters are entered. This number adjusted to make the length of a trailer record exactly 13 characters. This means that when a block is freed, the file pointer can be moved backwards 13 characters to determine the length and status of the physically adjacent lower block in the file. Physically adjacent free blocks (on either side of a block newly declared free) are combined into a larger block when possible.

FLINK and BLINK are initialized to 0. ***UA.ROVER*** and ***UA.AVAIL*** are initialized to the address of the first free block on the freelist. No trailer is written for the last free block on the freelist. This is done so that an accurate bytecount can be obtained for the UNITS file. It depends on the last free block also being the last block on the UNITS file.

The format of a reserved block is as follows:

Header Record:
TAG = R
SIZE = N
Unit Slots:
A list of SLOT records (described above).
Trailer Record:
TAG = R
SIZE = N
DUMMY = nZZZ

A roving pointer (***UA.ROVER***) through the free list is used to allocate blocks on a first-fit basis. (The use of a roving pointer prevents a buildup of small blocks at the beginning of the free list.) The first 10 characters of the UNITS file give the address of the first block on the free list.

When a block is allocated, the unneeded portion is returned to the free list unless it is

too small. The minimum usable block size is determined by the variable ***UA.MINGOODBLOCKSIZE***. Initially a file is set up with one free block of ***UA.BLOCKSIZE*** characters (currently set up to be 512*512*7, the maximum number of characters that can be addressed without resorting to a multiple level page table).

As indicated above, slot information is stored on the UNITS file described above. Information about the relations, groups, and disk address for the slot information on the UNITS file is stored on the RELATIONS file. This information is stored as a series of ***RELREC*** records with the following fields:

***NAME***: The name of the unit.

***DISKADDR***: The disk address of the slot information on the UNITS file.

***GEN***: The generalization of the unit.

***SPEC***: A list of the specializations of the unit.

***PROTO***: The prototype of the unit.

***INST***: A list of the instances of the unit.

***CLASS***: A list of the groups of which the unit is a member.

### 6.3 Implementation of Inheritance

One of the general advantages for inheritance of properties is that procedures and data structures can be declared at one unit and shared by its progeny without actually duplicating the structures in memory. For deep trees and large shared structures the space savings can be considerable. The obvious implementation and mental model for implementing such sharing involves chaining back through ancestors until the shared structure is found. In such an implementation, if a value is not in a unit, one must chain back (potentially all the way to the ***ROOT***) to either find the shared value or to determine that there is no such value. The time required for retrieval in this approach is proportional to the depth of the subtree. Offsetting the time disadvantage of this approach is the ease of updating values. To change a value one need only change the value in the most general ancestor and the effect of the change is automatically propagated to all the progeny with no further computational effort.

The UNIT Package uses a mechanism that offers more efficient retrieval with only a modest increase in memory usage and difficulty in updating values.

#### 6.3.1 Standard Format

When a slot is first defined in a UNIT Package knowledge base, it is created in ***Standard*** format. The unit in which this takes place is termed the ***top-level*** or most general unit for that slot. Slot fields are filled in as follows:

NAME: Name of the slot
DATATYPE: Name of the unit for the datatype.
ROLE: List of tokens; one of S, R, O, and U.
VALUE: Value (representation depends on datatype).
DEFAULT: (like value).
FIELDS: Optional fields.

For units that are specializations of this top-level unit, the datatype and role fields are used differently in certain situations. (These changes of format are internal conventions for the UNIT Package to minimize usage of space. They are invisible to a user.) The following defines three special formats.

### 6.3.2 S Format

NAME: Name of the slot.
DATATYPE: Name of Top-level unit for the slot.
ROLE: Atom S.
VALUE: NIL
DEFAULT: NIL
FIELDS: Optonal fields.

The idea behind this format is to speed up the retrieval of inherited values without copying them. Thus when a request is received (by the function GETFIELD [Section 5.6.5]) to retrieve a value of a slot which has been inherited, the request is deferred to the top-level unit. In terms of functions (described below), this is done using INHERITEDSLOT? (Section 5.5.10) which returns the name of the top-level unit from the datatype field if the role is *S*. This approach avoids tracing back up the tree as in the simple implementation without making multiple copies of the fields in all of the units involved.

In practice, the top-level unit contains all of the information for the role *S* slot and all of its progeny have the slot in *S* format. Any inquiry about the value field in the slot in progeny causes the corresponding value in the top-level slot to be returned. Any attempt to change a value in progeny results in an error message. To update the value in the top-level unit is simple and no processing is required to propagate the effects of the change to progeny. The slots in progeny are called *inherited* slots.

### 6.3.3 No change format

For *R*, *O*, and *U* roles, the information in the value and default fields is potentially changeable. However, up to the moment when this information is changed, it is advantageous to avoid making copies of the various slot fields in progeny.

The *No Change* (*NC*) format works essentially the same way as the *S* format. The top-level unit is in *Standard* format and its progeny start out in *NC* format. Any requests for value retrieval in the progeny cause the corresponding field in the top-level unit to be

returned. These progeny are said to be *tracking* the top-level, or *master* unit. Any change in the value of the top-level unit is thus automatically propagated to the progeny that are in ***NC*** format.

NAME: Name of the slot.
DATATYPE: Name of unit from which fields are to be copied.
ROLE: Atom NC
VALUE: NIL
DEFAULT: NIL
FIELDS: Optional fields

As soon as any field is changed in one of the progeny, the slot format is changed immediately to ***Standard*** format. All of the fields (except of course the changed one) are copied from the top-level unit. The slot is no longer an *unchanged* slot. It becomes a *master* slot for unchanged slots below it. The unchanged slots below are left in ***NC*** format, but the symbolic pointer in the datatype field is modified to point to the new *master* unit.

The ***NC*** format provides a mechanism to avoid copying fields as long as possible but still preserving the ability for change. It also implements the notion that the restrictions in ***R*** and ***O*** slots actually follow the changes in their top-level (or next level) units until they are changed. Once those slots have been changed, they are considered to be *further restricted* and changes that occur in the top-level unit are not propagated.

### 6.3.4 Role Change Format:

This is the format for a slot to which a CHANGEROLE operation has been performed (where the role has been changed to ***S***). (See Section 5.6.10.)

ROLE: List of tokens including S and either R or O.
DATATYPE: Just as in Top-Level Units.
VALUE: Just as in Top-Level Units.
DEFAULT: Just as in Top-Level Units.
FIELDS: Just as in Top-Level Units.

For slots having an ***R*** or ***O*** role which are later marked as ***S***, the role ***S*** is simply added to the role list. The progeny will then have the slot in the ***S*** format (above) with the datatype pointing to the unit to which the ***S*** role was added. In practice this role change causes a slot to switch to ***Standard*** format if it is in ***NC*** format. All progeny of the slot are then placed in ***S*** format.

# Chapter 7
## UNIT Package: Global Variables

Some variations in global variables may exist from site to site. When in doubt, check with your local UNIT Package programmer.

**COMMON.NOPRINTSLOTS**: (***ANCHOR***, ***CO-REFS***, and ***NOT-CO-REFS***) The names of slots that are not printed with a description or indefinite unit as the value of a slot.

**COREFILELIST**: A list of the names of the files included in the UNIT Package.

**EX?FILE**: The name of a file on which UNIT Package usage information is to be recorded.

**HOSTNAME**: The name of the host computer (e.g., SUMEX). This is only necessary if your site is not on the ARPAnet (i.e., (HOSTNAME) returns NIL).

**PROMPT**: The prompt character used by UE (default is ":").

**UA.AVAIL**: File pointer to first free block in the UNITS file.

**UA.BLOCKSIZE**: Initial size of the free block (512*512*7 bytes).

**UA.BUMPFLG**: Set by GCTRP break when fewer than ***UA.GCTRPCOUNT*** free list cells remain.

**UA.BUMPFN**: The name of the function that sorts units for moving to disk.

**UA.BUMPN**: Unit slots are paged out ***UA.BUMPN*** units at a time when more space is needed. This is done until ***UA.GCTRPCOUNT*** list cells are recovered or there are no more units whose slots are memory-resident. If ***UA.BUMPN*** is less than 1, it is interpreted as a fraction of the number of units currently in memory.

**UA.ERRMSG**: Text of the error message.

**UA.ERRNO**: Error number returned for error messages.

**UA.ERRTOKEN**: The unit, slot, field, relation, function, role, group, or file thought to be in error (or a list of several of these).

**UA.FILENAME**: Name of current knowledge base (no extension).

**UA.FIXED-FIELDS**: List of permanent fields in each slot.

**UA.GCTRPCOUNT**: The minimum number of list cells that must be free if unit slots are not to be paged out to disk from memory.

**UA.HASHFILE**: Name of the current HASH file (including extension).

**UA.HEADERPOS**: File pointer to the position for the header record of the block being written.

**UA.INCORELIST**: List of all units whose slots are in in memory.

**UA.MINGOODBLOCKSIZE**: Minimum acceptable free block size in the UNITS file. No smaller free block will be created. Can be adjusted to suit the environment where the **UNIT Package** is being used.

**UA.MSGFLG**: Flag that controls printing of messages by the **UNIT Package** itself. Initially it is NIL for suppression of messages. If set to T, the **UNIT Package** will print tracing messages to the terminal in addition to the normal error messages when error conditions are noticed. These messages are not intended to be of use to a normal user. They are intended to be helpful to programmers in debugging the **UNIT Package**.

**UA.NONUSERS**: Names of directories that do not correspond to users but where some unit information is stored.

**UA.PFLG**: T if unit paging is enabled.

**UA.RELFILE**: Name of the current RELATIONS file (including extension).

**UA.ROVER**: Roving file pointer to free blocks in the UNITS file.

**UA.SIZE-FACTOR**: Size multiplier when allocating blocks.

**UA.SIZE-INCR**: Size increment when allocating blocks. (See UA-PICKSIZE for formula.)

**UA.SYSNAME**: Name of the system for SYSTAT purposes (e.g., UNITS).

**UA.TIMESTAMP**: Current relative *time* for unit slot access. Incremented every time the slots of a different unit are accessed (i.e., a unit different from the last unit whose slots were accessed).

**UA.TRACEFLG**: When set to T, various trace messages are displayed on the primary output device as UNIT operations proceed.

**UA.TRAILPOS**: File pointer to the position for the trailer record of the block being written.

**UA.UNIT**: Name of the unit whose slots have been most recently accessed.

**UA.UNITFILE**: Name of the current UNITS file (including extension).

**UA.UNITLIST**: The list of all units in the current knowledge base.

**UA.UNITSIN**: Total number of units whose slots have been loaded into memory from disk for the current knowledge base (see below).

**UA.UNITSOUT**: Total number of units whose slots have been paged out to disk from memory for the current knowledge base (see below).

**UA.UPFLG**: T if unit paging information (above) is to be maintained.

**UA.USERS**: List of directories to be searched by NETWORK? when returning a list of knowledge bases.

**UEBEENCALLED**: T if UE-TOP has been called.

**UEBREVITY**: Profile flag. T if ***BRIEF*** mode is in effect.

**UEDEFAULTFLG**: Profile flag. T if default fields are to be printed.

**UEDEFN**: Profile flag. T if prompting for definitional roles is to occur.

**UEFIRSTLISPFLG**: T the first time the user calls **INTERLISP** from **UE**.

**UEHACKER**: Profile flag. T for ***HACKER*** mode.

**UEPROFILEFLG**: T if a user profile exists.

**UEQUIETFLG**: Profile flag. T if ***QUIET*** mode is in effect.

**UERECDEPTH**: UE recursion level.

**UERECORDFILE**: Name of the file in which a recording is being written.

**UETRANSMITFLG**: Profile flag. T if values are to be transmitted to progeny.

**UETYPEFLG**: Profile flag. T if prompting for unit type (instance or specialization) is to occur when a new unit is created.

**UEWITHFLG**: Profile flag. T if ***DESCR*** and ***INDEFINITE*** units are to be printed in ***WITH*** notation.

**UM.USERNAME**: Name of the current user. This variable is used by GETUSERNAME which provides the user's name for various bookkeeping operations. It's value is a string equal to the user's name (usually last name).

**UNITS.STORAGE**: A list of data types and their associate initial MINFS settings.

**UT.NETWORKS**: An association list of knowledge bases paired with units and functions that have been transferred from them to **UA.FILENAME**.

**UTNETWORK**: The current transfer knowledge base.

# Chapter 8

# UNIT Package: Internal Functions

## 8.1 Unit Access Internal Functions

### 8.1.1 UA-ASSIGNBLOCK (SIZE)

UA-ASSIGNBLOCKSIZE assigns the next free block of SIZE bytes on the UNITS file and returns its file pointer as a value.

### 8.1.2 UA-BUMPUNITS (CURUNIT)

UA-BUMPUNITS is called from UA-GETSLOT and UA-MAKEREL if ***UA.PFLG*** and ***UA.BUMPFLG*** are set. UA.BUMPFLG gets set when there is a GCTRP break (not enough free list cells left). UA-BUMPUNITS pages out ***UA.BUMPN*** of the memory-resident units according to the order specified by the function given by ***UA.BUMPFN***. CURUNIT is the unit whose slots are currently being accessed. UA-BUMPUNITS will not page its slots out to disk.

### 8.1.3 UA-CHANGEROLE (SLOT UNIT ROLE CASE MASTER)

UA-CHANGEROLE is a recursive subroutine of CHANGEROLE that propagates a change of inheritance role to ROLE for SLOT in UNIT to progeny. MASTER is the unit to be tracked. A number of CASEs are recognized:

ADD (Add an S role)
S (Convert to S format)
RO (Change role from R to O or vice versa)
US (Change role from U to S)
SRO (Convert S role to R or O)
NC (Convert to NC format)
CHANGE (Change role in top-level unit)
COPYROLE (Copy role and datatype fields, leaving others intact
[also used for CHANGE])

### 8.1.4 UA-COMISSUE (COMMAND JUNKFILE)

UA-COMISSUE issues COMMAND (a command string) to the operating system through a SUBSYS. Output goes to JUNKFILE if set or to the system null file (e.g., NUL: for TOPS-20).

### 8.1.5 UA-DELETEROLE1 (SLOT UNIT ROLE CASE)

UA-DELETEROLE1 is a recursive subroutine of DELETEROLE. It deletes ROLE from SLOT in UNIT and its progeny. Several CASEs are recognized:

RS (Delete S role from R and S or O and S)
NC (Convert from S format to NC format)
DELETE (Delete ROLE)
CLEAR (For unchanged and inherited slots. Just copy the role and datatype fields from master [which has already been changed])

### 8.1.6 UA-DELREL (UNIT)

UA-DELREL undefines UNIT.

### 8.1.7 UA-DELSLOT (SLOT UNIT KEEPFLG MSGFLG)

UA-DELSLOT is a recursive subroutine for DELETESLOT. It deletes SLOT from UNIT and its progeny according to KEEPFLG. If MSGFLG is T, then a ***DELETE*** message is sent to the slot as appropriate.

### 8.1.8 UA-ERRMSG (ERRNO TOKEN)

UA-ERRMSG prints the error message associated wth ERRNO for the specific TOKEN in error. The complete list of error numbers, token types, and messages is shown in Appendix C.

### 8.1.9 UA-GETLISPFILE (FILE)

UA-GETLISPFILE reads INTERLISP functions from FILE.

### 8.1.10 UA-GETREL (UNIT RELATION)

UA-GETREL returns the value of RELATION for UNIT.

### 8.1.11 UA-GETRELFILE (RELFILE)

UA-GETRELFILE reads the set of unit relations from RELFILE.

### 8.1.12 UA-GETRELREC (UNIT FILE)

UA-GETRELREC reads the relation record for UNIT from FILE. If UNIT is NIL, then UA-GETRELREC reads the next relation record.

### 8.1.13 UA-GETSLOT (SLOT UNIT)

UA-GETSLOT returns the value of SLOT of UNIT. If UNIT is not the most recently referenced unit, then UA-GETSLOT sets UTIMESTAMP property for UNIT, and resets ***UA.UNIT*** to UNIT. If ***UA.BUMPFLG*** is T, then calls UA-BUMPUNITS to page out the slots of some units to gain space. If the slots of UNIT are not in memory then calls UA-LOADUNIT to page them in. When accessing the value of a slot with UA-GETSLOT, it must be remembered that subsequent calls to UA-GETSLOT (or SLOT?) for another unit may cause the slots of any other original unit to be paged out.

### 8.1.14 UA-GETSLOTFIELD (SLOT UNIT FIELD)

UA-GETSLOTFIELD returns the value of FIELD of SLOT of UNIT, where FIELD can be one of VALUE, ROLE, DATATYPE, DEFAULT and FIELDS.

### 8.1.15 UA-HIERORDER (UNIT FLG)

UA-HIERORDER returns a list of slots in UNIT in hierarchical order. See SORTSLOTS (Section 5.5.5) for a description. FLG is either H or R.

### 8.1.16 UA-INITGLOBALVARS (FILE)

UA-INITGLOBALVARS initializes ***UA.FILENAME*** to FILE. It also initializes ***UA.UNITLIST***, ***UA.INCORELIST***, ***UA.TIMESTAMP***, ***UA.UNITSIN***, ***UA.UNITSOUT***, and ***UA.BUMPFLG***.

### 8.1.17 UA-LOADUNIT (UNIT)

UA-LOADUNIT loads the slots of UNIT into memory.

### 8.1.18 UA-LOCALFILENAME (FILENAME)

UA-LOCALFILENAME returns FILENAME minus any <USERNAME>. For example, <STEFIK>FOO.BAZ is converted to FOO.BAZ.

### 8.1.19 UA-MAKEREL (UNIT)

UA-MAKEREL defines UNIT as a unit.

### 8.1.20 UA-MAKESLOT (SLOT UNIT ROLE DATATYPE MASTER)

UA-MAKESLOT is a recursive subroutine for MAKESLOT. It creates slots at lower levels in hierarchy and patches existing slots encountered along the way.

### 8.1.21 UA-MAKESTANDARD (SLOT UNIT MASTER)

UA-MAKESTANDARD converts SLOT in UNIT from NC to standard format.

### 8.1.22 UA-OKSLOT? (SLOT UNIT ROLE DATATYPE)

UA-OKSLOT? is a subroutine of MAKESLOT that determines whether the DATATYPE and ROLE of an existing SLOT in UNIT conflict with a proposed slot.

### 8.1.23 UA-OPENFILE (FILENAME TYPE)

UA-OPENFILE opens FILENAME. TYPE is either RELATIONS or UNITS.

### 8.1.24 UA-OPENUNITFILE (UNITFILE)

UA-OPENUNITFILE opens UNITFILE as a UNITS file and initializes ***UA.AVAIL*** and ***UA.ROVER***.

### 8.1.25 UA-PACKAGEFN? (FN)

UA-PACKAGEFN? returns T if FN is a function in any of the ***UNIT Package*** files that have been loaded into memory.

### 8.1.26 UA-PATCHDELSLOT (SLOT UNIT OLDMASTER)

UA-PATCHDELSLOT is a recursive subroutine of DELETEUNIT used to patch slots in the progeny of a deleted unit that are tracking a slot in the deleted unit.

### 8.1.27 UA-PATCHRENAME (SLOT UNIT OLDMASTER NEWMASTER)

UA-PATCHRENAME is a recursive subroutine of RENAMEUNIT used to patch slots in the progeny of a renamed unit that are tracking a slot in the renamed unit.

### 8.1.28 UA-PICKSIZE (SIZE)

UA-PICKSIZE computes a free block size based on SIZE bytes for the UNITS file. (Slightly larger blocks than strictly required are allocated to allow for future expansion of a unit.)

### 8.1.29 UA-PROGENYSLOT? (SLOT UNIT)

UA-PROGENYSLOT? is a recursive subroutine of PROGENYSLOT?

### 8.1.30 UA-PUTHEADER (FILEPTR TAG SIZE FLINK BLINK)

UA-PUTHEADER writes the header for a block on the UNITS file.

### 8.1.31 UA-PUTREL (UNIT RELATION RELATIVE)

UA-PUTREL defines RELATION between UNIT and RELATIVE.

### 8.1.32 UA-PUTRELFILE (RELFILE FLG)

UA-PUTRELFILE writes current relations to RELFILE. If FLG is T then a new version of the file is written.

### 8.1.33 UA-PUTSLOTFIELD (SLOT UNIT FIELD VALUE)

UA-PUTSLOTFIELD puts VALUE into FIELD of SLOT of UNIT, where FIELD is one of: NAME, VALUE, ROLE, DATATYPE, DEFAULT, and FIELDS.

### 8.1.34 UA-PUTTRAILER (FILEPTR TAG SIZE)

UA-PUTTRAILER writes the trailer for a block on the UNITS file.

### 8.1.35 UA-PUTUNIT (UNIT)

UA-PUTUNIT writes the slots of UNIT to the UNITS file if they have been modified since they were last written to disk. (Allocates a new block for it if the old one won't fit.)

### 8.1.36 UA-PUTUNITFILE (UNITFILE)

UA-PUTUNITFILE writes modified units to UNITFILE, updates the Free List pointer, and closes the file.

### 8.1.37 UA-RELEASEBLOCK (PTR)

UA-RELEASEBLOCK returns the free block at address PTR to the UNITS file free list.

### 8.1.38 UA-RELEASEBLOCK1 (PTR HEADER)

UA-RELEASEBLOCK1 is a subroutine of UA-RELEASEBLOCK.

### 8.1.39 UA-RENAMESLOT (SLOT UNIT NEWNAME)

UA-RENAMESLOT is a recursive subroutine for RENAMESLOT. It renames SLOT in UNIT and its progeny to NEWNAME.

### 8.1.40 UA-SETDATATYPE (SLOT UNIT DATATYPE)

UA-SETDATATYPE changes the datatype of SLOT in UNIT and its progeny to be DATATYPE.

### 8.1.41 UA-SETROLE (SLOT UNIT ROLE MASTER)

UA-SETROLE is a subroutine of PUTROLE. It propagates ROLE for SLOT in UNIT to all progeny.

### 8.1.42 UA-STORERELREC (REC)

UA-STORERELREC defines a unit with properties specified in the relation record REC. It returns the name of the unit.

### 8.1.43 UA-TIMESORT (UNITLIST)

UA-TIMESORT orders units on UNITLIST according to recency of access (least recent accesses at the beginning of the list). It returns a sorted list of units.

### 8.1.44 UA-UNRELATE (UNIT RELATION RELATIVE)

UA-UNRELATE removes RELATIVE for UNIT from RELATION.

## 8.2 Unit Management Internal Functions

### 8.2.1 GETUSERNAME ()

GETUSERNAME returns the name of the current user. It first checks UM.USERNAME, then the LOGIN name (unless a member of ***UA.NONUSERS***), then the CONNECTED name (unless a member of ***UA.NONUSERS***), finally asks the user.

### 8.2.2 NOTEST (EVALUE ERESTRICTION ESLOT EUNIT EARG5)

NOTEST always returns T. It is intended for use by an attached procedure for checking something in cases where the test is not meaningful.

### 8.2.3 NILTEST (EVALUE ERESTRICTION ESLOT EUNIT EARG5)

NILTEST always returns NIL. It is intended for use by an attached procedure for checking something in cases where the test is not meaningful.

### 8.2.4 UM-CLEARBADVALUES (SLOT UNIT RESTRICTION)

UM-CLEARBADVALUES is a recursive subroutine of CLEARBADVALUES used to clear the VALUE field of SLOT in the progeny of UNIT that do not satisfy a given restriction.

### 8.2.5 UM-NEED-TV? (SLOT UNIT ROLE)

UM-NEED-TV? returns T if a terminal value is required for SLOT in UNIT, given the specification of ROLE.

## Appendix A

## UE Command Summary

| COMMAND | Examples | Effect |
|---|---|---|
| CONSISTENCY | CON | Check consistency of knowledge base |
| COPY | COP A B | Copy unit A to B |
| CREATE | CR B A SP | Create unit B as a specialization of A |
| DELETE | DE A | Delete unit A |
| DISPLAY | DI ROOT SP 2 | Display the generalization hierarchy starting at ROOT. Show class nodes (specializations) only and limit printing to 2 levels |
| DONE | DO | Return to operating system (OK works too) |
| EDIT | E A | Edit the slots in unit A |
| GROUP | G | Enter Group Command interpreter. (Operations on groups of units) |
| INSTANTIATE | I A | Make an instance out of A (a specialization) |
| LISP | LI | Invoke INTERLISP userexec (OK brings you back) |
| MSG | MS A B C | Send a B message to unit A (Starts up the procedure in the B slot with argument C) |
| ?MSGS | ?M A | List the tokens for procedures in unit A |
| MATCH | MA | Print list of units matching description |
| MOVE | MO A | Move unit A to another place in the generalization hierarchy |
| NETWORK | N | Switch to another knowledge base |
| PRINT | P A | Print slots in unit A |
| | P A B | Print B slot in unit A |
| !PRINT | !P A | Same as PRINT with profile set to VERBOSE HACKER NO-WITH-NOTATION |
| RECORD | REC A | Make recording of this session in file A. A subsequent REC command turns it off. |
| RENAME | REN A B | Rename unit A to B |
| SAVE | SA | Save your KB but do not exit. |
| SET-PROFILE | SE BR HA OK | Set Profile to BRIEF HACKER |
| SHOWREFS | SH A | Show units whose slots reference unit A |

| | | |
|---|---|---|
| SPECIALIZE | SPE A | Change Instance node A to a class node (specialization) |
| SPLITUNIT | SPL A | Introduce a new node between A and its parent |
| SUMMARYFILE | SU | Create a summary of the KB |
| TRANSFER | TR | Get units from another KB |
| TSHOW | TS | Display units and functions that have been transferred to the current KB |
| WHATSNEW | WH 7 | List units that have been changed in the past 7 days. |

---

NOTES:

You can type ? or HELP during any of the commands for an explanation of the required parameters. If you just type a command name, you will be prompted for each of the arguments in turn. Many of the commands have options that would take to long to list here. If you explore the commands with the ?, you will find the options.

---

| Character | Function |
|---|---|
| ↑E | (<ctrl>-E) Cancel a Command |
| ↑P | (<ctrl>-P) Look--Get trace of where you are. |
| ↑K | (<ctrl>-K) Recur the Network editor. |
| ↑O | (<ctrl>-O) Stop this output. |

## Appendix B

## Slot Editor Command Summary

| COMMAND | Examples | Effect |
|---|---|---|
| CDEFAULT | CD A | Clear the default of slot A |
| CLEARVALUE | CLE A | Clear the value of slot A |
| CLASSIFY | CL | Change the group classification of this unit. |
| COPY | CO A B C | Copy value of slot A from slot C in unit B. |
| CREATE | CR A LIST O | Create slot A of datatype LIST and inheritance role O. |
| DELETE | DE A | Delete slot A. |
| DISPLAY | DI | List the names of the slots in this unit. |
| DONE | DO | Return to UE. (OK works too.) |
| EDIT | E A | Edit slot A with datatype editor. |
| FIELD-EDIT | F A | Edit the fields in slot A. |
| !HELP | !H | Display this message |
| MSG | M A B | Send a B message to slot A. (Starts up the procedure in the A field of the slot or in the A slot of the associated datatype.) |
| PRINT | P B | Print slot B. |
| !PRINT | !P B | Same as PRINT command with the profile set to VERBOSE HACKER NO-WITH-NOTATION |
| RENAME | R A B | Rename slot A to B. |
| SETDEFAULT | SE A | Set the DEFAULT of slot A. |
| SHOWRELATIONS | SH | Show the built-in relations of this unit (e.g., GEN, PROTO) |
| SORT | SO H | Sort slots in hierarchical order |

# Appendix C

## Error Messages

The complete list of UNIT Package error messages is shown below.

| Error Number | Error Token | Error Message |
|---|---|---|
| 1 | FILENAME | FILE not found. |
| 2 | UNIT | UNIT not a Specialization. |
| 3 | UNIT | INSTANCE cannot have progeny. |
| 4 | UNIT1 UNIT2 | UNIT1 is not an ancestor of UNIT2. |
| 5 | FILENAME | NETWORK must be closed. |
| 6 | UNIT | UNIT slots not in memory--can't be paged out. |
| 7 | UNIT | UNIT slots have no disk address. |
| 9 | UNIT | Relation record not found for UNIT. |
| 10 | UNIT | UNIT not defined. |
| 11 | UNIT | UNIT already defined. |
| 12 | UNIT | UNIT has no Generalization. |
| 13 | UNIT | Illegal name for new UNIT. |
| 16 | UNIT | Instances of Primitive or Slot UNIT are not units. |
| 17 | UNIT | Proposed Primitive or Slot UNIT has instances that are units. |
| 18 | UNIT | Datatype UNIT must be a member of PRIMITIVE group. |
| 19 | UNIT | UNIT not an Instance. |
| 20 | SLOT UNIT | SLOT not found in UNIT. |
| 21 | SLOT UNIT | SLOT already defined in UNIT. |
| 22 | SLOT UNIT | Illegal name for new SLOT in UNIT. |
| 23 | SLOT UNIT | Illegal change to directly Inherited S Role SLOT in UNIT. |
| 25 | SLOT UNIT | Operation illegal for non Top-level SLOT in UNIT. |
| 26 | SLOT UNITs | Conflict: SLOT already defined in Progeny UNITs. |
| 30 | ROLE SLOT UNIT | ROLE not found for SLOT In UNIT. |
| 31 | ROLE SLOT UNIT | Illegal ROLE for SLOT in UNIT. |
| 32 | ROLE SLOT UNIT | Incompatible ROLE already set for SLOT in UNIT. |
| 40 | CLASS UNIT | Invalid CLASS for UNIT. |
| 50 | SLOT UNIT | Extra SLOT in Slotlist of UNIT. |
| 51 | SLOT UNIT | Missing SLOT in Slotlist of UNIT. |
| 52 | SLOT UNIT | Duplicate SLOT in Slotlist of UNIT. |
| 60 | UNIT | Illegal change to the ROOT UNIT. |
| 71 | FIELD SLOT UNIT | Illegal operation on Permanent FIELD of SLOT in UNIT. |

| | | |
|---|---|---|
| 72 | FIELD SLOT UNIT | FIELD not defined for SLOT in UNIT. |
| 73 | FIELD SLOT UNIT | FIELD already defined for SLOT in UNIT. |
| 74 | FIELD SLOT UNIT | Illegal name for FIELD of SLOT in UNIT. |
| 80 | RELATION UNIT | Illegal RELATION for UNIT. |
| 92 | FUNCTION | LISP FUNCTION not defined. |
| 110 | SLOT UNIT | LISP error from Attached Procedure in SLOT of UNIT. |
| 130 | ROLE SLOT UNIT | Terminal Value required for ROLE of SLOT in UNIT. |
| 150 | SLOT UNIT | Definition conflict for SLOT in UNIT. |

## References


[Bobrow, 1977]
D. G. Bobrow and T. Winograd, An Overview Of KRL, A Knowledge Representation Language. *Cognitive Science*, Vol. 1, No. 1, January 1977, pp. 3-46.

[Brachman, 1977]
R. J. Brachman, What's In A Concept: Structural Foundations For Semantic Networks. *International Journal of Man-Machine Studies*, Vol. 9, 1977, pp. 127-152.

[Dahl, 1966]
O-J. Dahl and K. Nygaard, SIMULA - An ALGOL-based Simulation Language. *CACM*, Vol. 9, No. 9, September 1966, pp. 671-677.

[Friedland, 1979]
P. E. Friedland, *Knowledge-Based Experiment Design In Molecular Genetics*. STAN-CS-79-771 (HPP-79-29), Dept. of Computer Science, Stanford University, October 1979.

[Goldberg, 1976]
A. Goldberg and A. Kay, *SMALLTALK-72 Instruction Manual*. SSL 76-6, Xerox Palo Alto Research Center, 1976.

[Hendrix, 1975]
G. G. Hendrix, Expanding The Utility Of Semantic Networks Through Partitioning. *IJCAI4*, 1975, pp. 115-121.

[Knuth, 1968]
D. E. Knuth, *The Art Of Computer Programming. Volume 1: Fundamental Algorithms*. Reading, Mass.: Addison-Wesley, 1968

[Quillian, 1968]
M. R. Quillian, Semantic Memory. In M. Minsky (Ed.), *Semantic Information Processing*. Cambridge, Mass.: MIT Press, 1968.

[Stefik, 1979]
M. J. Stefik, An Examination Of A Frame-Structured Representation System. *Proceedings of the Sixth International Joint Conference On Artificial Intelligence*, August 1979, pp. 845-852.

[Stefik, 1980]
M. J. Stefik, *Planning With Constraints*. STAN-CS-80-784 (HPP-80-2), Dept. of Computer Science, Stanford University, January 1980.

[Teitelman, 1978]
W. Teitelman, *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, Ca., October 1978.

[Woods, 1975]
W. A. Woods, What's In A Link: Foundations For Semantic Networks. In D. G. Bobrow and A. Collins (Eds.), *Representation And Understanding: Studies In Cognitive Science*. New York: Academic Press, 1975.

UNCLASSIFIED

Security Classification

**DOCUMENT CONTROL DATA - R & D**

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATING ACTIVITY

DEFENCE RESEARCH ESTABLISHMENT ATLANTIC

2a. DOCUMENT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP

3. DOCUMENT TITLE

UNIT PACKAGE USER'S GUIDE

4. DESCRIPTIVE NOTES (Type of report and inclusive dates) TECH MEMO

5. AUTHOR(S) (Last name, first name, middle initial)

SMITH, REID G., & FRIEDLAND, PETER

| 6. DOCUMENT DATE | 7a. TOTAL NO. OF PAGES | 7b. NO. OF REFS |
|---|---|---|
| DECEMBER 1980 | 108 | 12 |

| 8a. PROJECT OR GRANT NO. | 9a. ORIGINATOR'S DOCUMENT NUMBER(S) |
|---|---|
| | D.R.E.A. TECHNICAL MEMORANDUM 80/L |
| 8b. CONTRACT NO. | 9b. OTHER DOCUMENT NO.(S) (Any other numbers that may be assigned this document) |

10. DISTRIBUTION STATEMENT

| 11. SUPPLEMENTARY NOTES | 12. SPONSORING ACTIVITY |
|---|---|
| | Department of National Defence, Canada, Research and Development Branch. National Science Foundation (Grant MC578-02777) |

13. ABSTRACT

The UNIT Package is a frame-structured, hierarchically-organized knowledge representation and acquisition system. The package has nodes for individuals, classes, indefinite individuals, and descriptions. Links between the nodes are structured with explicit definitional roles, types of inheritance, defaults, and various datatypes.

This report contains an overview of the UNIT Package, a guide to the use of the Unit Editor (UE), a summary of the contents of the BOOTSTRAP knowledge base, and a summary of high-level access functions.

It also contains implementation information, including a discussion of the underlying data structures, global variables and low-level functions.

DSIS
78-070

UNCLASSIFIED

Security Classification

**KEY WORDS**

ARTIFICIAL INTELLIGENCE (AI)
KNOWLEDGE REPRESENTATION
REPRESENTATION LANGUAGES
KNOWLEDGE ACQUISITION
UNITS
FRAMES
SEMANTIC NETWORKS
KNOWLEDGE BASES
KNOWLEDGE-BASED SYSTEMS

**INSTRUCTIONS**

1. ORIGINATING ACTIVITY: Enter the name and address of the organization issuing the document.

2a. DOCUMENT SECURITY CLASSIFICATION: Enter the overall security classification of the document including special warning terms whenever applicable.

2b. GROUP: Enter security reclassification group number. The three groups are defined in Appendix 'M' of the DRB Security Regulations.

3. DOCUMENT TITLE: Enter the complete document title in all capital letters. Titles in all cases should be unclassified. If a sufficiently descriptive title cannot be selected without classification, show title classification with the usual one-capital-letter abbreviation in parentheses immediately following the title.

4. DESCRIPTIVE NOTES: Enter the category of document, e.g. technical report, technical note or technical letter. If appropriate, enter the type of document, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the document. Enter last name, first name, middle initial. If military, show rank. The name of the principal author is an absolute minimum requirement.

6. DOCUMENT DATE: Enter the date (month, year) of Establishment approval for publication of the document.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the document.

8a. PROJECT OR GRANT NUMBER: If appropriate, enter the applicable research and development project or grant number under which the document was written.

8b. CONTRACT NUMBER: If appropriate, enter the applicable number under which the document was written.

9a. ORIGINATOR'S DOCUMENT NUMBER(S): Enter the official document number by which the document will be identified and controlled by the originating activity. This number must be unique to this document.

9b. OTHER DOCUMENT NUMBER(S): If the document has been assigned any other document numbers (either by the originator or by the sponsor), also enter this number(s).

10. DISTRIBUTION STATEMENT: Enter any limitations on further dissemination of the document, other than those imposed by security classification, using standard statements such as:

    (1) "Qualified requesters may obtain copies of this document from their defence documentation center."

    (2) "Announcement and dissemination of this document is not authorized without prior approval from originating activity."

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document, even though it may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall end with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (TS), (S), (C), (R), or (U).

    The length of the abstract should be limited to 20 single-spaced standard typewritten lines; 7½ inches long.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a document and could be helpful in cataloging the document. Key words should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context.

DEFENCE RESEARCH ESTABLISHMENT ATLANTIC